Universidade Estadual de Campinas
Instituto de Computação

Leandro Negri Zanotto

High Performance Collision Cross Section (HPCCS) HPC Techniques to Accelerate the Collision Cross Section Calculation

High Performance Collision Cross Section (HPCCS) Utilização de Técnicas de HPC para Aceleração do Cálculo da Seção de Choque Transversal

CAMPINAS
2019

Leandro Negri Zanotto

High Performance Collision Cross Section (HPCCS) HPC Techniques to Accelerate the Collision Cross Section Calculation

High Performance Collision Cross Section (HPCCS) Utilização de Técnicas de HPC para Aceleração do Cálculo da Seção de Choque Transversal

Dissertação apresentada ao Instituto de Computação da Universidade Estadual de Campinas como parte dos requisitos para a obtenção do título de Mestre em Ciência da Computação.

Thesis presented to the Institute of Computing of the University of Campinas in partial fulfillment of the requirements for the degree of Master in Computer Science.

Supervisor/Orientador: Prof. Dr. Guido Costa Souza de Araujo
Co-supervisor/Coorientador: Prof. Dr. Gabriel Heerdt

Este exemplar corresponde à versão final da Dissertação defendida por Leandro Negri Zanotto e orientada pelo Prof. Dr. Guido Costa Souza de Araujo.

CAMPINAS
2019

Ficha catalográfica
Universidade Estadual de Campinas
Biblioteca do Instituto de Matemática, Estatística e Computação Científica
Silvania Renata de Jesus Ribeiro - CRB 8/6592

Z17h

Zanotto, Leandro Negri, 1979-
High Peformance Collision Cross Section (HPCCS) - HPC techniques to accelerate the collision cross section calculation / Leandro Negri Zanotto. – Campinas, SP : [s.n.], 2019.

Orientador: Guido Costa Souza de Araújo.
Coorientador: Gabriel Heerdt.
Dissertação (mestrado) – Universidade Estadual de Campinas, Instituto de Computação.

1. Programação paralela (Computação). 2. Computação de alto desempenho. 3. Mobilidade iônica. I. Araújo, Guido Costa Souza de, 1962-. II. Heerdt, Gabriel, 1987-. III. Universidade Estadual de Campinas. Instituto de Computação. IV. Título.

Informações para Biblioteca Digital

Título em outro idioma: High Performance Collision Cross Section (HPCCS) - Utilização de técnicas de HPC para aceleração do cálculo da seção de choque transversal
Palavras-chave em inglês:
Parallel programming (Computer science)
High performance computing
Ion mobility
Área de concentração: Ciência da Computação
Titulação: Mestre em Ciência da Computação
Banca examinadora:
Guido Costa Souza de Araújo [Orientador]
Fábio Cesar Gozzo
Marcio Machado Pereira
Data de defesa: 13-12-2019
Programa de Pós-Graduação: Ciência da Computação

Identificação e informações acadêmicas do(a) aluno(a)
- ORCID do autor: https://orcid.org/0000-0003-1335-4812
- Currículo Lattes do autor: http://lattes.cnpq.br/4161931644468407

Universidade Estadual de Campinas
Instituto de Computação

Leandro Negri Zanotto

High Performance Collision Cross Section (HPCCS) HPC Techniques to Accelerate the Collision Cross Section Calculation

High Performance Collision Cross Section (HPCCS) Utilização de Técnicas de HPC para Aceleração do Cálculo da Seção de Choque Transversal

Banca Examinadora:

- Prof. Dr. Guido Souza Costa de Araújo (Supervisor/ Orientador)
  Institute of Computing - UNICAMP
- Prof. Dr. Fabio Cesar Gozzo
  Institute of Chemistry - UNICAMP
- Dr. Marcio Machado Pereira
  Institute of Computing - UNICAMP

A ata da defesa, assinada pelos membros da Comissão Examinadora, consta no SIGA/Sistema de Fluxo de Dissertação/Tese e na Secretaria do Programa da Unidade.

Campinas, 13 de dezembro de 2019

Agradecimentos

Agradeço primeiramente a minha família, por estarem sempre ao meu lado em todos os momentos da minha vida. Agradeço-lhes por toda ajuda, apoio e auxílio fazendo com que meus objetivos fossem atingidos. Também gostaria de agradecer o meu orientador e co-orientador por tudo que me ensinaram durante esses anos trabalhando juntos a fim de concretizar essa dissertação de mestrado. Foram de grande importância em todo o ciclo de aprendizado. Por último agradeço o Professor Dr. Munir Skaf do Instituto de Química da Unicamp que me proporcionou esse desafio ao trabalhar nesse projeto. Ao Dr. Paulo Cesar Telles de Souza pela ajuda quando precisei duranto o desenvolvimento do projeto. À Fundação de Amparo à Pesquisa do Estado de São Paulo (FAPESP), processos № 2012/24750-6, 2013/08293-7, 2016/04963-6 pelos recursos computacionais, e ao Centro de Computação em Engenharia e Ciências pelas instalações e ao Instituto de Computação e Unicamp por permitir que eu realizasse o mestrado contribuindo com meu aprendizado.

Resumo

A técnica de Mobilidade Iônica junto com a Espectrometria de Massa (IM-MS) tem sido utilizada desde 2003 por laboratórios de pesquisa e análises, quando foram introduzidos os primeiros equipamentos comerciais. Ela é usada como uma ferramenta de separação molecular, técnica cromatográfica e também para obter informação estrutural de íons moleculares. A interpretação dos dados obtidos ainda é um desafio, dependendo dos cálculos da seção de choque transversal (CCS) contra um gás de arraste. Este trabalho, apresenta um novo software, *High Performance Collision Cross Section* - HPCCS, que, baseado no método de trajetória, realiza os cálculos de CCS utilizando técnicas de *High Performance Computing* como paralelização, vetorização e otimização. Agora é possível calcular o CCS de maneira eficiente, desde para pequenas moléculas orgânicas até proteínas complexas com um número maior de átomos. Os resultados mostraram que, comparados com o software usado atualmente (MOBCAL), houve um ganho em média de 78 vezes em um nó de um cluster com 24 cores e 48 threads, utilizando Simultaneous Multithreading (SMT).

Abstract

Ion Mobility coupled to Mass Spectrometry technique (IM-MS) have been used since 2003 for research and analysis laboratories, when they were commercially introduced. It has been used as a tool for molecular separation, chromatography technique, and to obtain structural information for molecular ions. The interpretation of the resulting data is still a challenge, depending on collision cross section (CCS) calculation against a buffer gas. This work, presents a new software, High Performance Collision Cross Section - HPCCS, which is based on the trajectory method, using High Performance Computing techniques like parallelization, vectorization and optimization. By using HPCCS now calculate the CCS efficiently, from small organic molecules to protein complexes with a larger number of atoms. The results presented in this work when comparing to the state of the art software (MOBCAL), show an average speedup of 78 times on a cluster node with 24 cores and 48 threads, with Simultaneous Multithreading (SMT).

List of Figures

List of Tables

Contents

Chapter 1

Introduction

Mass spectrometry (MS) is an indispensable analytical tool in many related fields of science, like medicine. It is employed for example to explore single cells or objects from outer space, as a way to elucidate unknown substances. It has also been commonly used in forensics, quality control of drugs, foods and polymers analysis [7].

The interest in ion mobility spectrometry (IMS), when ion mobility is coupled with mass spectrometry, is increasing since it presents an effective means of separating gaseous ions working as a chromatography technique. In IMS an electric field forces the ions to drift along a path, thorough a countercurrent inert gas atmosphere, whereby they are separated due to their mobility [7]. IMS can separate isobaric ions of different charge state, resulting from their distinct speed of propagation along the electric field of the ion mobility tube, or distinguish isobars of the same charge state by their steric properties [7].

The rotationally-averaged collision cross-section represents the effective area for the interaction between an individual ion and the neutral gas, through which it is traveling. CCS is an important distinguishing characteristic of an ion, related to its chemical structure and three-dimensional conformation [7].

Mobcal is an important software, widely used, for theoretical CCS calculation. It is based on three different treatments of the ion-buffer gas collisions: the projection approximation (PA), the exact hard sphere scattering (EHSS), and the trajectory method (TM). TM is the most accurate method, being the best choice for CCS estimates for highly charged macromolecules, such as proteins and proteins complexes [20]. Theoretical computations of CCS for biomacromolecular systems, under TM approximation, are inefficient with Mobcal, because of its outdated program language and technology, thus limiting its usage to studies of small proteins and organic molecules.

High Performance Computing (HPC) explores the computational resources like novel VLSI technology, parallelization algorithms and computer architectures to enable the solution of complex Engineering and Scientific applications in a feasible time. The processors are improving each year, some features like larger cache for faster data access, multicores for parallel processing and larger vector units to process four add instructions at once are examples of such improvements. These resources are available in clusters, cloud and even in desktops or mobile devices. The time to process the results is reduced, but to achieve it, parallel algorithms are necessary to make an efficient usage of the new hardware [9].

High Performance Collision Cross Section - HPCCS, is a new software proposed in this dissertation, capable of performing CCS calculation for a large variety of molecular ions, ranging from small organic molecules to large protein complexes containing tens to hundreds thousand of atoms. It uses current processors features, like multicore processing and vectorization. It is based on Mobcal, and focused on the Trajectory Method. When the original Mobcal was written, processors were single core, so the CCS calculation was sequential [26].

For parallelization HPCCS used OpenMP[4], which is a well-known API available for C/C++ and Fortran languages, used to parallelize code blocks using shared memory model. MPI[6] which uses distributed memory to split the code block into process across the cluster using more than one cluster node.

Some loops were fused to execute the code in one single loop using only the necessary variables. The vectorization was necessary to speedup the loops using one single core. Having the execution faster than MOBCAL on a single core, the code when parallelized had a good speedup comparing to MOBCAL which runs only using a serial execution.

Using the ideas proposed in this dissertation, HPCCS was totally rewritten and improved with new features that enable application speedups resulting in the HPCCS program which can tackle large biomolecules within much shorter times and higher accuracy, thus helping the interpretation of experimental IM-MS data [26].

Chapter 2

Background

There is an analytical interest for compound identification from molecular masses, by mass spectrometry technique, as a mechanism to enable the mapping of the constituents of complex mixtures. Fields of application of MS are: Physics, Radiochemistry, Geochemistry, Organic chemistry, Polymer chemistry, Biochemistry, Physical chemistry, Thermochemistry, Quality control, Environmental analysis, Petroleum chemistry, Food chemistry, Biomedical studies, Material sciences, Field portable MS, Space missions, Military applications, medicine, etc.

2.1 Experimental Model – Mass Spectrometry and Ion Mobility

The basic principle of mass spectrometry is to generate ions, from either inorganic or organic compounds, by a suitable method, separating these ions by their mass-to-charge ratio (m/z) and detecting them qualitatively and quantitatively. The sample may be ionized thermally, by electric fields or by impacting on energetic electrons, ions or photons. Ions can be single ionized atoms, clusters, molecules or their fragments or associates. Ion separation can be affected by static or dynamic electric or magnetic fields [22].

As demonstrated with great success by the time-of-flight analyzer, ion separation by m/z can also be effected in field-free regions, providing the ions with well-defined kinetic energy at the entrance of flight path. Figure 2.1 shows a general scheme of any mass spectrometer. Often, several types of samples inlets are attached to the ion source housing. Transfer of the sample from atmospheric pressure to the high vacuum of the ion source and mass analyzer is accomplished by a vacuum lock. Mass spectrometry is an ideal detection method for gaseous ions eluting from an ion mobility device, making the ion mobility-mass coupling receive attention since the 1970s [8].

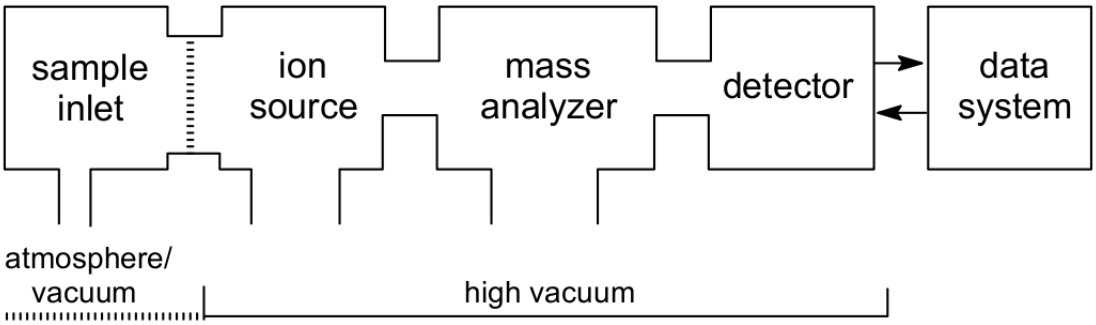

Figure 2.1: General scheme of any mass spectrometer.

Recently, IMS technology has been used, for example, to detect illegal substances in airport scanner devices. On portable devices it is extremely effective and useful to obtain molecular information, such for small molecules, proteins and even viruses [12].

The ion mobility separation is described as a gas-phase electrophoresis technique, whereby gaseous ions are separated according to their size, shape, and charge in the presence of an electric field. An inert buffer gas (i.e. helium or nitrogen) fills the drift tube at low vacuum pressures or at atmospheric pressure conditions. Ions move according to diffusion process through the drift tube as the energies of the ions are similar to the thermal energy of the buffer gas. Ions will have different mobilities in a given drift tube device, which allows the separation of ion mixtures by their time of flight (TOF), related to collision cross section (CCS), an structural information to be obtained [12].

The most widespread developed and employed approach on IMS is the Drift-Time Ion Mobility Spectrometry (DTIMS), wich is the only IMS method providing a direct measure of CCS based in ion mobilities. Figure 2.2 shows a drift tube instrument, filled with inert buffer gas in a counter direction of the ion motion. The weak electric field applied to the drift tube is generated using a series of resistors and a DC potential [12].

DTIMS does not work with continuous injection of ions, therefore packets of ions are introduced into the drift tube using an ion gate or ion funnel. Ion packets can range in width from 100 μs to 200 μs, because of the use of ion packets. After the ions are injected into the drift tube, the species begin to separate based on their mobility through the buffer gas. Ions which have more elongated conformations will undergo more collisions with the buffer gas, taking a longer time to drift through the tube than more compact structures. Figure 2.2 presents these concepts [12].

2.1.1 Simulation Models

The mobility of gas phase ion is a measure of how rapidly it moves through a buffer gas, under the influence of an electric field. The mobility depends on the average collision cross section, which it turn depends on the geometry [16].

Mobcal is a software developed by Shvartsburg and Jarrold, from Indiana University, to calculate the theoretical CCS based on input coordinate files similar to Protein Data Base file, derived from X-ray crystallography, NMR studies or MD simulations. It uses three different methods to CCS calculation: Projection Approximation (PA), Exact Hard Sphere Scattering (EHSS) and Trajectory Method (TM), each one is described below.

Projection Approximation

CCS is determined by averaging the area of projections on a plane, considering all possible orientations by rotations. However, this method ignores the long-distance interactions and the scattering process between the ion and buffer gas [25].

The calculation using PA is fast since it ignores the scattering process and long-range interactions between the ion and the gas [**?**].

Exact Hard Sphere Scattering

The EHSS method calculates CCS by averaging the momentum transfer cross section over the relative velocity and collision geometry. It takes into account scattering and collision processes, but does not consider the effects of long range interactions. In summary, it is a simplification of the trajectory method, explained below [27]. It is commonly used on structural proteomics to examine them due to their large number of atoms.

Trajectory Method

TM is regarded as the most reliable and accurate method. It combines all the effects, including scattering events, long-range interactions and multiple collisions. The only weakness to consider is that time consuming, specially for macromolecules ions [24].

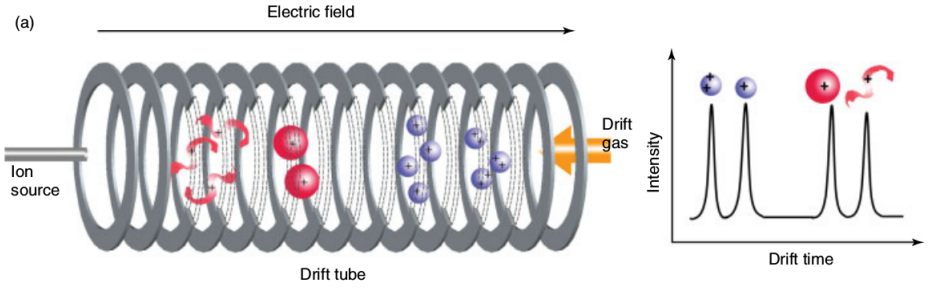

Figure 2.2: Scheme of a drift tube, filled by isobaric ions and being separated by their different chemical characteristics

Chapter 3

Related works

Mobcal is the main software to calculate the Collision Cross Section (CCS). It has been used since 1996 and its modules are described bellow explaining how it works.

3.1 Mobcal

Developed by Shvartsburg and Jarrold to calculate the theoretical CCS based on input coordinate file, called mfj, derived from X-ray crystallography, NMR studies or MD simulations [20]. Each section bellow describes the entire software flow. The Figure 3.4 is an example with the 10 first lines of Ubiquitin (PDB id: 1UBQ). Bellow will be presented the Mobcal execution flow. Note that all executions are serial and no compiler optimization was made since the code breaks using them.

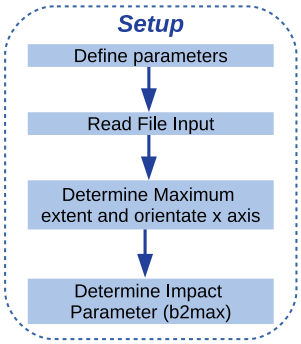

Figure 3.1: The Mobcal Setup Flow

Figure 3.1 starts defining the parameters, reads the .mfj input file, determines the maximum extent and orientation axis, having the molecule in the desired position for collision with the gas buffer, so the collisions will cover almost the total molecule area and finally the impact parameter will be calculate.

List 3.1 shows how the ***gsang*** module is called. It uses three loops that calculate the velocity $\boldsymbol{v}$ and $\boldsymbol{b}$. The parameter $\boldsymbol{b}$ is used to determine the initial gas position related to the impact parameter. It also rotates the molecule randomly for each trajectory, trying to cover the entire molecule; these parameters are used in the ***gsang*** module.

Listing 3.1: Mobility calculation

```
 do 4011 ig=1,inp
       q1st(ig)=0.d0
       q2st(ig)=0.d0
 4011 continue
c
       do 4040 ic=1,itn
       if(ip.eq.1) write(*,681) ic
  681 format(/1x,'cycle number, ic =',i3)
       om11st(ic)=0.d0
       om12st(ic)=0.d0
       om13st(ic)=0.d0
       om22st(ic)=0.d0
       do 4010 ig=1,inp
       gst2=pgst(ig)*pgst(ig)
       v=dsqrt((gst2*eo)/(0.5d0*mu))
       if(ip.eq.1) write(*,682) ic,ig,gst2,v
  682 format(/1x,'ic =',i3,1x,'ig =',i4,1x,'gst2 =',1pe11.4,
     ?1x,'v =',e11.4)
       temp1=0.d0
       temp2=0.d0
c
       do 4000 im=1,imp
       if(ip.eq.1.and.im.eq.1) write(*,683)
  683 format(/5x,'b/A',8x,'ang',6x,'(1-cosX)',4x,'e ratio',4x,'
     theta',
     ?7x,'phi',7x,'gamma')
       rnb=xrand()
       call rantate
       bst2=rnb*b2max(ig)
       b=ro*dsqrt(bst2)
c
       call gsang(v,b,erat,ang,d1,istep)
       hold1=1.d0-dcos(ang)
       hold2=dsin(ang)*dsin(ang)
       if(ip.eq.1) write(*,684) b*1.d10,ang*cang,hold1,erat,
     ?theta*cang,phi*cang,gamma*cang
  684 format(1x,1pe11.4,7(e11.4))
       temp1=temp1+(hold1*b2max(ig)/dfloat(imp))
       temp2=temp2+(1.5d0*hold2*b2max(ig)/dfloat(imp))
 4000 continue
```

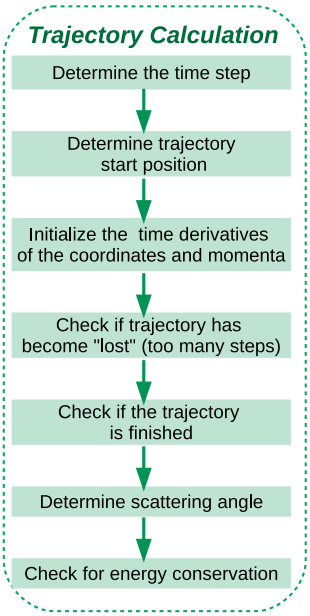

Figure 3.2: The Mobcal Trajectory Flow

Figure 3.2 presents the steps to calculate the trajectory (***gsang*** module). Only the parameters calculated before and passed to the module change at each execution. As discussed later in this dissertation this code can be vectorized, but at the time it was coded (1996), there were no processors capable for enabling that.

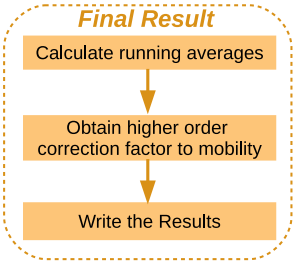

Figure 3.3: The Mobcal Result Flow

Figure 3.3 represents the flow followed by Mobcal for the average cross-section area calculations. It calculates the scattering angles, with each angle corresponding to a point in Monte Carlo integration. In the end, it uses the same three loops to integrate the averages [23].

3.1.1 Input Reading

Mobcal used an input file with .mfj extension to describe the molecule which the program will read and calculate its CCS. This file is presented on Figure 3.4 and the file lines are described below:

- First line is the file label, it could be any text to identify it.
- Second line represents the number of coordinate sets.
- Third line is the total number of atoms.
- Fourth line is the unit considered for the x, y and z axes, in this case the unit ang represents angstrom, au for atomic units.
- Fifth line tells how Mobcal will use the charges or not. If the label is "equal" it will specify uniform charge distribution, if "none" it will specify no charge, or "calc" to specify a non-uniform charge distribution.
- Sixth line is a correction factor for coordinates, usually 1.0000, based on ion relaxation during scattering events.
- Seventh line is divided into columns the x, y, z, integer mass and partial charge, to be used in case "calc" is set.

```
04_1ubq
1
1235
ang
calc
1.0000
29.95   27.43   19.18   14      0.1592
28.88   28.41   19.4    12      0.0221
29.39   29.6    20.21   12      0.6123
30.15   29.44   21.16   16      -0.5713
```

Figure 3.4: Mobcal's Input file of Ubiquitin with the first ten lines.

3.1.2 Impact Parameter Calculation

The impact parameter, usually denoted by the letter $\boldsymbol{b}$, of a trajectory is defined to be the closest distance to the origin the particle would achieve if it moved in the straight line determined by its initial velocity [21]. Mobcal assumes fully elastic collisions and rigid non rotating ions in which kinetic energy is conserved. In Mobcal the ***b2max*** computes the maximum impact parameter value for each trajectory in order to obtain ion extension.

Listing 3.2: b2max Calculation

```
c      determine b2max
       dbst2=1.d0
       dbst22=dbst2/10.d0
       cmin=0.0005
       if(im2.eq.0) write(*,652) cmin
  652  format(//1x,'set up b2 integration - integration over',
      ?' impact parameter',//1x,
      ?'minimum value of (1-cosX) =',1pe11.4,/)
       do 3030 ig=inp,1,-1
       gst2=pgst(ig)*pgst(ig)
       v=dsqrt((gst2*eo)/(0.5d0*mu))
       ibst=dint(rmaxx/ro)-6
       if(ig.lt.inp)ibst=dint(b2max(ig+1)/dbst2)-6
       if(ibst.lt.0) ibst=0
       if(ip.eq.1) write(*,650) gst2,v
  650  format(/1x,'gst2 =',1pe11.4,1x,'v =',e11.4,/6x,'b',
      ?10x,'bst2',7x,'X ang',7x,'cos(X)',6x,'e ratio')
 3000  bst2=dbst2*dfloat(ibst)
       b=ro*dsqrt(bst2)
       call gsang(v,b,erat,ang,d1,istep)
       cosx(ibst)=1.d0-dcos(ang)
       if(ip.eq.1) write(*,651) b,bst2,ang,cosx(ibst),erat
  651  format(1x,1pe11.4,6(1x,e11.4))
       if(ibst.lt.5) goto 3010
       if(cosx(ibst).lt.cmin.and.cosx(ibst-1).lt.cmin.and.
      ?cosx(ibst-2).lt.cmin.and.cosx(ibst-3).lt.cmin.and.
      ?cosx(ibst-4).lt.cmin) goto 3020
 3010  ibst=ibst+1
       if(ibst.gt.500) then
       write(*,653)
  653  format(1x,'ibst greater than 500')
       close (8)
       stop
       endif
       goto 3000
 3020  b2max(ig)=dfloat(ibst-5)*dbst2
 3040  b2max(ig)=b2max(ig)+dbst22
       b=ro*dsqrt(b2max(ig))
       call gsang(v,b,erat,ang,d1,istep)
       if(1.d0-dcos(ang).gt.cmin) goto 3040
 3030  continue
       if(im2.eq.0) then
```

```
      write(*,637)
  637 format(/5x,'gst',11x,'b2max/ro2',9x,'b/A',/)
      do 3050 ig=1,inp
 3050 write(*,630) pgst(ig),b2max(ig),ro*dsqrt(b2max(ig))*1.0d10
  630 format(1x,1pe11.4,5x,e11.4,5x,e11.4)
      endif
```

3.1.3 Trajectories Method Calculation

To obtain structural information from ion-mobility measurements, cross sections can be calculated for conformers obtained from experiments or generated by molecular modeling methods, then compared with experimental CCS results. The collision cross section gives an orientationally averaged result and the mobility, K, of molecular ion in a low pressure buffer gas within the linear response regime can be calculated from [20] [24].

$$K = \frac{\sqrt{18\pi}}{16}\left[\frac{1}{m}+\frac{1}{m_B}\right]^{\frac{1}{2}}\frac{ze}{(k_BT)^{\frac{1}{2}}}\frac{1}{\Omega_{avg}^{(1,1)}}\frac{1}{N} \tag{3.1}$$

where m is the molecular ion mass, m_B and N is the buffer gas mass and density, respectively, ze is the ionic charge and $\Omega_{avg}^{(1,1)}$ is the orientationally averaged collision integral. Equation 3.1. Since the ω integral is related to the scattering angles χ, Figure 3.5, the orientationally averaged collision cross section can be accurately determined by integrating over all possible collision geometries [20] [24].

$$\begin{aligned}\Omega_{avg}^{(1,1)} = \frac{1}{8\pi^2}\int_0^{2\pi}d\theta\int_0^{\pi}d\phi\sin\phi\int_0^{2\pi}d\gamma\frac{\pi}{8}\left(\frac{\mu}{k_BT}\right)^3\int_0^{\pi}dg\ e^{-\frac{\mu g^2}{2k_BT}}g^5 \\ \times\int_0^{\infty}db2b(1-cos\chi(\theta,\phi,\gamma,g,b))\end{aligned} \tag{3.2}$$

where μ is the reduced mass, g is the magnitude of the relative collision velocity, and b the impact parameter for all possible ion orientations defined by the three Euler angles θ, ϕ and γ. Equation 3.2 can be calculated by numerical integration of the trajectories against buffer gas particles, toward the molecular ion propagated in an intermolecular interaction potential V [3].

$$\begin{aligned}V(\theta,\phi,\gamma,b,r) = 4\sum_i^n\varepsilon_i\left[\left(\frac{\sigma_i}{r_i}\right)^{12}-\left(\frac{\sigma_i}{r_i}\right)^{6}\right]-\frac{\alpha}{2}\left(\frac{ze}{n}\right)^2 \\ \left[\left(\sum_i^n\frac{x_i}{r_i^3}\right)^2+\left(\sum_i^n\frac{y_i}{r_i^3}\right)^2+\left(\sum_i^n\frac{z_i}{r_i^3}\right)^2\right]\end{aligned} \tag{3.3}$$

The first therm is the sum of Lennard-Jones (6-12) potential between the buffer gas and individual atom i of the molecular ion. The Lennard-Jones parameters, ε (depth) and σ (distance) are provided in the literature. The second term is the charge-induced dipole interaction with α being the gas polarizability and x_i, y_i, z_i and r_i the relative position of each individual atom i. For properly computing the interactions between ion and N_2 gas, there additional terms are required in Equation 3.3: an ion-quadrupole interaction and the orientation of the linear N_2 molecule during the collision process [3]. This approach to the

CCS calculation is called the Trajectory Method (TM), providing an accurate prediction for the cross section of a given candidate geometry, although it is computationally very expensive.

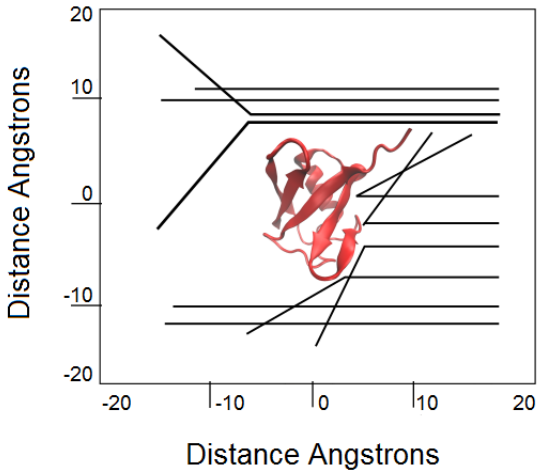

Figure 3.5: The accurate estimation of CCS requires the calculation of all possible collision angles between a buffer gas and a target molecule.

3.2 IMPACT

In 2015, Benesch and co-workers published the Ion Mobility Projection Approximation Calculation Tool (IMPACT) [17], which provides CCS estimates within the PA framework at low computational costs. PA method allow CCS calculations for systems containing a large number of atoms, but results can be highly inaccurate [1]. This method does not consider the interactions between gas and molecular ion.

For small molecules using PDB files IMPACT is a good choice to calculate their CCS values. Nevertheless, although it is faster than other programs for protein complexes, it is not accurate. The experimental results section 5.5 compares the values of IMPACT, Mobcal and HPCCS.

IMPACT uses only PDB for its computation, and since this format file does not have the charge, so it does not differentiate molecules by their charge.

A suggestion IMPACT makes is using it on the fly on MD simulations. Each simulation step will produce a result since this software is fast enough to not interfere on the simulation time.

Chapter 4

High Performance Collision Cross Section - HPCCS

Computers have become essential due to their ability to perform calculations, visualizations and general data processing at an incredible ever-increasing speed. Some decades ago the computers processors were built to run the software in serial using one processor with a single core. To accelerate the calculation transistor size was shirinked, thus enabling faster clock speeds and adding some other features like vector units, pipeline and cache.

This continuous density increase in VLSI technology finally hit a thermal barrier due to the difficulty in draining heat out of the silicon die. To handle the barrier problem, the transistors continued to shrink, but the frequency did not scale anymore and more cores were added to compensate for that. Today the computers are built with two cores at least on desktops or mobile devices, capable to do parallel processing. On clusters there are more than one processor per node, with much more cores on each one for scientific computation [9].

High Performance Computing (HPC) is a set of techniques that includes large scale computer cluster and parallelization and optimizations techniques that are used to deliver the performance that a single desktop cannot do, so as to solve large problems in science, engineering or business. Such machines process large amount of data, producing big data for analysis and calculations what would take months to have the results on a single computer.

As the computer architecture has evolved, the old softwares can be rewritten to use the current features and run faster. Mobcal is the most cited software to process CCS. Since it was written in 1996 its execution is only serial and does not accept any compiler optimization, so only the CCS of small molecules can be calculated. Driven by this limitation and by the availability of modern HPC clusters and techniques, we developed HPCCS. Bigger molecules like proteins can now be calculated in a feasible time even in multicore desktops.

Based on Mobcal trajectory method, HPCCS was re-written in the C/C++ computer language. The functions were modularized for better support. The input file is different from Mobcal, using a PQR file which follows a standard for molecule description. HPC techniques were used to execute using only the necessary amount of memory, parallelization was applied to make use of all available processor cores and vectorization was used to

speedup the potential calculation. The sections bellow present details of design HPCSS program.

First, HPCCS serial execution will be presented, explaining all modules and their improvements. A comparison will be made showing the adopted strategies to achieve the results. HPCCS was designed in two versions: one using OpenMP for shared memory on a single node and MPI + OpenMP version using multiples nodes.

4.1 Files Organization

Mobcal has only a single source code file with all modules. If a bug fix is necessary, it could be hard to identify and fix. On HPCCS the modules were splitted into files for better support. On Figure 4.1 the HPCCS files organization is presented.

In the ***config*** folder there are three input files, the AtomsMLJHe.csv, AtomsMLJN2.csv and config.in. Global and constant variables are inside the ***headers*** folder with the respective file names. The other files are the headers for the source files. The .pqr file stores the molecule input for CCS calculation. The .ccp files are the source code of the HPPCS implementation using the .pqr file as input, .csv as initial values for calculation and config.in as setup. There is a Make file to make the software compilation easier.

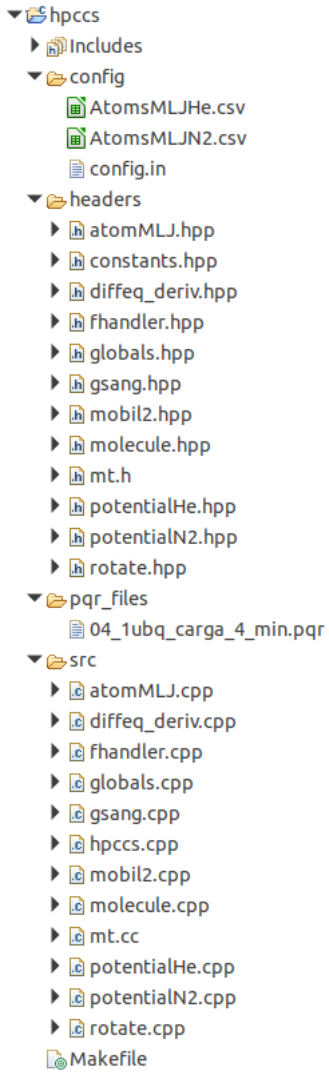

Figure 4.1: HPCCS Files Organization

4.2 Execution Flow

4.3 Input Files

There are three input files used on HPCCS, .csv, .in and .pqr. Each one is described bellow.

4.3.1 .CSV file

The .csv found at config folder; describes atom mass and its Lennard-Jones parameters (σ and ϵ) related to the gas, these information were hard coded on Mobcal. The idea was to put these values outside the code using it inside the .csv file for better support when other values are needed, the user can just replace the values on this file. File AtomsMLJN2.csv stores the values are related to N2, other gas is required, the user just needs to change the values and create a new file (e.g AtomsMLJCO2.csv) and change the code to read it.

Figure 4.2 presents the AtomsMLJHe Input file.

H	1.008	1.0414152645E-22	2.38E-010
He	4.0026	0	0
Li	6.941	0	0
Be	9.0122	0	0
B	10.811	0	0
C	12.01	2.1469176222E-022	3.043E-010
N	14.01	2.1469176222E-022	3.043E-010
O	16.00	2.1469176222E-022	3.043E-010
F	19.00	1.71436E-022	2.43E-010
Ne	20.1797	0	0
Na	22.99	4.45414E-024	3.5369E-010
Mg	24.305	0	0
Al	26.9815	0	0
Si	28.09	2.1629393955E-022	3.5E-010
P	30.9738	0	0
S	32.060	2.1629393955E-022	3.5E-010
Cl	35.453	0	0
K	39.0983	0	0
Ar	39.948	0	0
Ca	40.078	0	0
Sc	44.9559	0	0
Ti	47.867	0	0
V	50.9415	0	0
Cr	51.9961	0	0
Mn	54.938	0	0
Fe	55.850	2.1629393955E-022	3.5E-010
Ni	58.6934	0	0

Figure 4.2: AtomsMLJHe Input File

The AtomsMLJHe.csv file is read in **fhandler.cpp**, by means of function **fcoordinates**, presented on code which contains Listing 4.1. An object atomMLJArray is created to store all file information, it has an array for each column storing file row at each array position.

Listing 4.1: code snipet from fcoordinates function to read .csv files

```
1     if(gas == 2){
2         infile.open("config/AtomsMLJN2.csv");
3     }else {
4         infile.open("config/AtomsMLJHe.csv");
5     }
6     if (infile.is_open()){
7         while(getline(infile,string)){
8             istringstream sstream(string);
9             sstream >> atomName >> mass >> epsilon >> sigma;
10            atomMLJArray[i].setAtomName(atomName);
11            atomMLJArray[i].setMass(mass);
12            atomMLJArray[i].setSigma(sigma);
13            atomMLJArray[i].setEpsilon(epsilon);
14            i++;
15        }
16        infile.clear();
17        infile.close();
18    }
19
20    #if defined(__INTEL_COMPILER)
21    #pragma vector aligned
22    #endif
23    for (int atoms = 0; atoms < i; atoms++){
24        atomMLJArray[atoms].setEox4(4.0 * atomMLJArray[atoms].
              getEpsilon());
25        atomMLJArray[atoms].setRo6lj(pow(atomMLJArray[atoms].
              getSigma(),6));
26        atomMLJArray[atoms].setRo12lj(pow(atomMLJArray[atoms].
              getSigma(),12));
```

```
27          atomMLJArray[atoms].setDro6(6.0 * atomMLJArray[atoms].
               getRo6lj());
28          atomMLJArray[atoms].setDro12(12.0 * atomMLJArray[atoms].
               getRo12lj());
29      }
```

4.3.2 .IN file

This is the config file for some iof the parameters required to execute the CCS calculation. It stores the number of conformation, number of complete cycles for average mobility calculation (itn), number of points in velocity integration (inp) and number of points for the Monte Carlo integration of the impact parameter (imp), the temperature and a number to select the buffer gas (1 for Helium or 2 for Nitrogen).

4.4 Molecule Class

A Molecule class was created to represent the PQR file. There are two files, the header molecule.hpp and the implementation molecule.cpp. All fields were aligned in memory, user-level optimization performance that has been found effective for ultimate performance [11].

Vectorization works best when the data being consumed is contiguous in memory. It was used on each field of SoA (Structure of Arrays) to access the data with unit strides. AVX (Advanced Vector Extensions) instructions were used to vectorize the loops, by means of the auto vectorization found on optimization level 3.

Listing 4.2: Molecule Header File

```
1  #ifndef SRC_HEADERS_MOLECULE_HPP_
2  #define SRC_HEADERS_MOLECULE_HPP_
3
4  class Molecule {
5      public:
6          double *fx __attribute__((aligned(16)));
7          double *fy __attribute__((aligned(16)));
8          double *fz __attribute__((aligned(16)));
9          double *ox __attribute__((aligned(16)));
10         double *oy __attribute__((aligned(16)));
11         double *oz __attribute__((aligned(16)));
12         float *charge __attribute__((aligned(16)));
13         float *xmass __attribute__((aligned(16)));
14         float *eox4 __attribute__((aligned(16)));
15         double *ro6lj __attribute__((aligned(16)));
16         double *ro12lj __attribute__((aligned(16)));
17         double *dro6 __attribute__((aligned(16)));
18         double *dro12 __attribute__((aligned(16)));
19         //Constructor
20
21         Molecule(unsigned int size);
22         ~Molecule();
23 };
24
25 #endif /* SRC_HEADERS_MOLECULE_HPP_ */
```

Listing 4.3: Molecule Implementation

```
1
```

```
2 #include <headers/molecule.hpp>
3 #if defined(__INTEL_COMPILER)
4 #include <aligned_new>
5 #else
6 #include <new>
7 #endif
8 Molecule::Molecule(unsigned int size){
9              fx = new double[size];
10             fy = new double[size];
11             fz = new double[size];
12             ox = new double[size];
13             oy = new double[size];
14             oz = new double[size];
15             charge = new float[size];
16             xmass = new float[size];
17             eox4 = new float[size];
18             ro6lj = new double[size];
19             ro12lj = new double[size];
20             dro6 = new double[size];
21             dro12 = new double[size];
22         };
23
24         Molecule::~Molecule(){
25             delete[] fx;
26             delete[] fy;
27             delete[] fz;
28             delete[] ox;
29             delete[] oy;
30             delete[] oz;
31             delete[] charge;
32             delete[] xmass;
33             delete[] eox4;
34             delete[] ro6lj;
35             delete[] ro12lj;
36             delete[] dro6;
37             delete[] dro12;
38         };
```

4.4.1 PQR

A PQR file is a modified PDB file, including columns containing the per-atom charge (Q) and radius (R). PQR files are used in several computational biology and chemistry packages, specially in Molecular Dynamics simulations. This format is still amenable to visualization with standard molecular graphics programs. This file was chosen because PDB does not have the charges, information used in Trajectory Method. Table 4.4.1 describes the PQR file format.

Record Name	Serial	Atom Name	Residue Name	Residue Number	X	Y	Z	Charge	Radius
ATOM	1	N	GLY	1	42.19	19.23	38.04	0.2943	1.824
ATOM	2	C	GLY	1	43.42	19.71	37.38	-0.01	1.908
ATOM	3	C	GLY	1	43.07	20.55	36.16	0.6163	1.908
ATOM	4	O	GLY	1	42	20.35	35.6	-0.5722	1.6612
ATOM	5	H	GLY	1	42.42	18.62	38.82	0.1642	0.6

Table 4.1: Table describing the first 5 lines of 1MLT PQR File

The user can build this file using a service like PDB2PQR [5]. It is necessary to upload the PDB file, select the force field and select the pKa options, both related to charge assignment. There are some concerns about the file creation to get the result near the experimental one. Depending on which force field is used, the file charges will be slightly different. Everything that directly or undirectly affects these fields will result in a different CCS. Geometries are important to calculate CCS, minimization is not required in all cases, but for small ions the DFT optimization is recommended.

As Ion Mobility experiments are working on gas phase, the structures change slightly from crystal, explaining why minimization is important.

The target charge depends on the experiment. Having a data bank with lots of CCS values for proteins in specific charge state the charges can enable one to studied and to decide which one to use for a specific protein. If you are trying to reproduce some experiment, you can see the MS spectra to obtain the total charge. But if you want just to evaluate some ion, you can choose your target charge depending in what you want. To reproduce the behavior of proteins in blood, for example, use blood pH and then the protein charge in this pH.

Listing 4.4 presents the steps to read the PQR file and create the Molecule object which represents the file.

Listing 4.4: code snipet from fcoordinates function to read .csv files

```
1
2  /* Check the number of atoms */
3      i = 0;
4      infile.open (filename);
5              if (infile.is_open()){
6                  while(getline(infile,string)) // To get you all the
                       lines.
7                  {
8                      istringstream sstream(string);
9                      sstream >> recordName;
10
11                     if (recordName == "ATOM"){
12                         ++i;
13                     }
14                 }
15                 infile.close();
16             }
17
18     numberOfAtoms = i;
19     molecule = new Molecule(numberOfAtoms);
20
21     i = 0;
22     romax = 0.0;
23     infile.open(filename);
24     if (infile.is_open()){
25         while(getline(infile,string)) // To get you all the lines.
26         {
27             istringstream sstream(string);
28             sstream >> recordName >> serial >> atomName >>
                   residueName >> chainId >> x >> y >> z >> charges >>
                   radius;
29
30             if (recordName == "ATOM"){
31                 atomName.erase(std::remove_if(atomName.begin(),
                       atomName.end(), [](char x){return std::isdigit(x
                       );}), atomName.end());
32                 molecule->fx[i] = x;
33                 molecule->fy[i] = y;
```

```
34                  molecule->fz[i] = z;
35                  molecule->charge[i] = charges;
36                  for (j = 0; j < atomsMLJCount; j++){
37                      if (atomName.length() > 1 && isupper(atomName.
                           at(1))){
38                           atomName = atomName.at(0);
39                      }
40                      if (atomMLJArray[j].getAtomName() == atomName){
41                          molecule->xmass[i] = atomMLJArray[j].
                               getMass();
42                          molecule->eox4[i] = atomMLJArray[j].getEox4
                               ();
43                          molecule->ro6lj[i] = atomMLJArray[j].
                               getRo6lj();
44                          molecule->ro12lj[i] = atomMLJArray[j].
                               getRo12lj();
45                          molecule->dro6[i] = atomMLJArray[j].getDro6
                               ();
46                          molecule->dro12[i] = atomMLJArray[j].
                               getDro12();
47                          if(atomMLJArray[j].getSigma() > romax)
                               romax = atomMLJArray[j].getSigma();
48                          break;
49                      }
50                  }
51                  ++i;
52              }
53          }
54          infile.clear();
55          infile.close();
56          status = 2;
57          }else{
58              status = 1;
59          }
60
61          if (status != 1){
62          #if defined(__INTEL_COMPILER)
63          #pragma vector aligned
64          #endif
65          for (i = 0; i < numberOfAtoms; i++){
66              fxo += molecule->fx[i] * molecule->xmass[i];
67              fyo += molecule->fy[i] * molecule->xmass[i];
68              fzo += molecule->fz[i] * molecule->xmass[i];
69              m2 += molecule->xmass[i];
70          }
71
72          fxo /= m2;
73          fyo /= m2;
74          fzo /= m2;
75
76          #if defined(__INTEL_COMPILER)
77          #pragma vector aligned
78          #endif
79          for (i = 0; i < numberOfAtoms; i++){
80              molecule->fx[i] = (molecule->fx[i] - fxo) * 1.0e-10;
81              molecule->ox[i] = molecule->fx[i];
82              molecule->fy[i] = (molecule->fy[i] - fyo) * 1.0e-10;
83              molecule->oy[i] = molecule->fy[i];
84              molecule->fz[i] = (molecule->fz[i] - fzo) * 1.0e-10;
85              molecule->oz[i] = molecule->fz[i];
86          }
87          }
```

4.5 Rotate Module

The Rotate module is responsible for rotating the molecule on random positions for each trajectory simulation. Listing 4.5 will rotate the molecule to determine maximum extent and orientation along x axis, so the rotation will be internally. Listing 4.6 the fx, fy, fz are random values generated for the Merssene Twister algorithm, which uses the second rotate module at each simulation, to maximize the molecule side exposition to the gas for CCS calculation.

Listing 4.5 uses three loops to rotate the molecule, each one for each axis. Comparing to the other two 4.6 and 4.7 there is only one loop to rotate the three axis, transforming the three loops in one loop is called *loop fusion*, it was done since the range of three loops are the same and some variables can be reused.

Listing 4.5: Code to Rotate the Molecule on Mobcal

```
1  subroutine rotate
2  c
3  c      Rotates the cluster/molecule.
4  c
5        implicit double precision (a-h,m-z)
6        parameter (len=40000)
7        common/printswitch/ip,it,iu1,iu2,iu3,iv,im2
8        common/constants/mu,ro,eo,pi,cang,ro2,dipol,emax,m1,m2,
9       ?xe,xk,xn,mconst,correct,romax,inatom,icoord,iic
10       common/charge/pcharge(len)
11       common/coordinates/fx(len),fy(len),fz(len),
12      ?ox(len),oy(len),oz(len)
13       common/ljparameters/eolj(len),rolj(len),eox4(len),
14      ?ro6lj(len),ro12lj(len),dro6(len),dro12(len)
15       common/trajectory/sw1,sw2,dtsf1,dtsf2,cmin,ifail,ifailc,inwr
16       common/angles/theta,phi,gamma
17       common/xrandom/i1,i2,i3,i4,i5,i6
18 c
19       if(iu2.eq.1.or.iu3.eq.1) write(*,610) theta*cang,phi*cang,
20      ?gamma*cang
21    610 format(//1x,'coordinates rotated by ROTATE',//1x,
22      ?'theta=',1pe11.4,1x,'phi=',e11.4,1x,'gamma=',1pe11.4,/)
23 c
24       do 1000 iatom=1,inatom
25       rxy=dsqrt(ox(iatom)*ox(iatom)+(oy(iatom)*oy(iatom)))
26       if(rxy.eq.0.d0) goto 1010
27       otheta=dacos(ox(iatom)/rxy)
28       if(oy(iatom).lt.0.d0) otheta=(2.d0*pi)-otheta
29       ntheta=otheta+theta
30    1010 fx(iatom)=dcos(ntheta)*rxy
31    1000 fy(iatom)=dsin(ntheta)*rxy
32 c
33       do 2000 iatom=1,inatom
34       rzy=dsqrt(oz(iatom)*oz(iatom)+(fy(iatom)*fy(iatom)))
35       if(rzy.eq.0.d0) goto 2010
36       ophi=dacos(oz(iatom)/rzy)
37       if(fy(iatom).lt.0.d0) ophi=(2.d0*pi)-ophi
38       nphi=ophi+phi
39    2010 fz(iatom)=dcos(nphi)*rzy
40    2000 fy(iatom)=dsin(nphi)*rzy
41 c
42       do 3000 iatom=1,inatom
43       rxy=dsqrt(fx(iatom)*fx(iatom)+(fy(iatom)*fy(iatom)))
44       if(rxy.eq.0.d0) goto 3010
45       ogamma=dacos(fx(iatom)/rxy)
46       if(fy(iatom).lt.0.d0) ogamma=(2.d0*pi)-ogamma
47       ngamma=ogamma+gamma
48    3010 fx(iatom)=dcos(ngamma)*rxy
```

```
 3000 fy(iatom)=dsin(ngamma)*rxy
c
      if(iu2.eq.0) goto 4000
      write(*,620)
  620 format(9x,'initial coordinates',24x,'new coordinates',/)
      do 4020 iatom=1,inatom
 4020 write(*,600) ox(iatom),oy(iatom),oz(iatom),fx(iatom),
     ?fy(iatom),fz(iatom)
  600 format(1x,1pe11.4,2(1x,e11.4),5x,3(1x,e11.4))
      close (10)
c
 4000 return
      end
```

Listing 4.6: Code to Rotate the Molecule

```
#include <headers/constants.hpp>
#include <headers/globals.hpp>
#include "math.h"

void rotate(double rnt, double rnp, double rng, double *fx, double
    *fy, double *fz){

//  Rotates the cluster/molecule.
    double rxy, rtheta, rphi, rgamma;
    float ogamma = 0.0, ophi = 0.0, otheta = 0.0, ntheta = 0.0,
       ngamma = 0.0, nphi = 0.0;
    unsigned int iatom;

    rtheta = rnt * 2.0 * pi;
    rphi = asin((rnp * 2.0)-1.0) + (pi/2.0);
    rgamma = rng * 2.0 * pi;

    #if defined(__INTEL_COMPILER)
    #pragma vector aligned
    #endif
    #pragma omp simd
    for (iatom = 0; iatom < numberOfAtoms; iatom++){

        rxy = sqrt((molecule->ox[iatom] * molecule->ox[iatom]) + (
           molecule->oy[iatom] * molecule->oy[iatom]));
        if(rxy == 0.0) {
            fx[iatom] = cos(ntheta) * rxy;
            fy[iatom] = sin(ntheta) * rxy;
        } else {
            otheta = acos(molecule->ox[iatom] / rxy);
            if(molecule->oy[iatom] < 0.0){
                otheta = (2.0 * pi) - otheta;
            }
            ntheta = otheta + rtheta;
            fx[iatom] = cos(ntheta) * rxy;
            fy[iatom] = sin(ntheta) * rxy;
        }

        rxy = sqrt((molecule->oz[iatom] * molecule->oz[iatom]) + (
           fy[iatom] * fy[iatom]));
        if(rxy == 0.0){
             fz[iatom] = cos(nphi) * rxy;
             fy[iatom] = sin(nphi) * rxy;
        }else {
           ophi = acos(molecule->oz[iatom] / rxy);
           if(fy[iatom] < 0.0){
              ophi=(2.0 * pi) - ophi;
           }
            nphi = ophi + rphi;
            fz[iatom] = cos(nphi) * rxy;
            fy[iatom] = sin(nphi) * rxy;
        }
```

```
50
51          rxy = sqrt((fx[iatom] *  fx[iatom]) + (fy[iatom] *  fy[
                iatom]));
52          if(rxy == 0.0){
53            fx[iatom] = cos(ngamma) * rxy;
54            fy[iatom] = sin(ngamma) * rxy;
55          } else {
56             ogamma = acos(fx[iatom] / rxy);
57             if(fy[iatom] < 0.0){
58                 ogamma = (2.0 * pi) - ogamma;
59             }
60             ngamma = ogamma + rgamma;
61             fx[iatom] = cos(ngamma) * rxy;
62             fy[iatom] = sin(ngamma) * rxy;
63          }
64      }
65  }
```

Listing 4.7: Code to Rotate the Molecule for b2max calculation

```
1  void rotate(){
2
3  //  Rotates the cluster/molecule.
4      double ogamma, ophi, otheta, ntheta, ngamma, nphi, rxy;
5      double rnt, rnp, rng;
6      unsigned int iatom;
7
8      ophi = 0.0;
9      nphi = 0.0;
10     ngamma = 0.0;
11     ntheta = 0.0;
12
13     #if defined(__INTEL_COMPILER)
14     #pragma vector aligned
15     #endif
16     for (iatom = 0; iatom < numberOfAtoms; iatom++){
17
18         rxy = sqrt((molecule->ox[iatom] * molecule->ox[iatom]) + (
               molecule->oy[iatom] * molecule->oy[iatom]));
19
20         if(rxy == 0.0) {
21             molecule->fx[iatom] = cos(ntheta) * rxy;
22             molecule->fy[iatom] = sin(ntheta) * rxy;
23         } else {
24             otheta = acos(molecule->ox[iatom] / rxy);
25             if(molecule->oy[iatom] < 0.0){
26                 otheta = (2.0 * pi) - otheta;
27             }
28             ntheta = otheta + theta;
29             molecule->fx[iatom] = cos(ntheta) * rxy;
30             molecule->fy[iatom] = sin(ntheta) * rxy;
31         }
32
33         rxy = sqrt((molecule->oz[iatom] * molecule->oz[iatom]) + (
               molecule->fy[iatom] * molecule->fy[iatom]));
34
35         if(rxy == 0.0){
36             molecule->fz[iatom] = cos(nphi) * rxy;
37             molecule->fy[iatom] = sin(nphi) * rxy;
38         }else {
39             ophi = acos(molecule->oz[iatom] / rxy);
40             if(molecule->fy[iatom] < 0.0){
41                ophi=(2.0 * pi) - ophi;
42             }
43             nphi = ophi + phi;
44             molecule->fz[iatom] = (cos(nphi) * rxy);
45             molecule->fy[iatom] = (sin(nphi) * rxy);
46         }
```

```
47
48          rxy = sqrt((molecule->fx[iatom] * molecule->fx[iatom]) + (
                molecule->fy[iatom] * molecule->fy[iatom]));
49
50          if(rxy == 0.0){
51            molecule->fx[iatom] = (cos(ngamma) * rxy);
52            molecule->fy[iatom] = (sin(ngamma) * rxy);
53          } else {
54              ogamma = acos(molecule->fx[iatom] / rxy);
55              if(molecule->fy[iatom] < 0.0){
56                  ogamma = (2.0 * pi) - ogamma;
57              }
58              ngamma = ogamma + agamma;
59              molecule->fx[iatom] = (cos(ngamma) * rxy);
60              molecule->fy[iatom] = (sin(ngamma) * rxy);
61          }
62      }
63  }
```

4.6 Potential Calculation

For potential calculation between pair of atoms, the Lennard-Jones (LJ) potential is used. Since it is an accurate model, it produces trusty bond energies and bond lengths [13].

The distance between any two atoms is r. The energy and length are the properties of the gas which has the corresponding parameters ε and σ. The attractive interactions are due to overlap of the electron clouds which forces electrons into higher energy states [13].

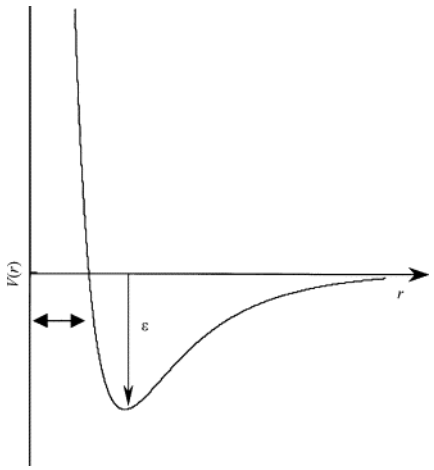

Figure 4.3: Lennard-Jones potential.

Mobcal uses function *dljpot* for potential calculation. According tothe total Figure 4.4 created by Intel Vtune Profiler 2019 u4, the *dljpot* function represents 97,62% of CPU elapsed time, once scaterring processes are consequence of interactions between atoms from protein and the buffer gas.

Elapsed Time: 106.023s

CPU Time:	792.880s
Total Thread Count:	8
Paused Time:	0s

Top Hotspots

This section lists the most active functions in your application. Optimizing these hotspot functions typically results in improving overall application performance.

Function	Module	CPU Time
dljpotHe	hpccs	771.095s
dljpotHe	hpccs	7.012s
dljpotHe	hpccs	3.985s
__ieee754_acos_fma	libm.so.6	2.732s
rotate	hpccs	2.478s
[Others]		5.577s

*N/A is applied to non-summable metrics.

Figure 4.4: CPU time consuming for Mobcal functions, obtained with Vtune Profiler.

Listing 4.8 shows the potential calculation on Mobcal. The interaction is between the drift gas, in this case the He, and each atom from the molecule. Depending on where it is called the function can perform many unnecessary calculations.

Listing 4.8: He Pontential Calculation on Mobcal

```
1         subroutine dljpot(x,y,z,pot,dpotx,dpoty,dpotz,dmax)
2 C
3 C       Subroutine to calculate L-J + ion-dipole potential.
4 C
5         implicit double precision (a-h,m-z)
6         parameter (len=40000)
7         common/printswitch/ip,it,iu1,iu2,iu3,iv,im2
8         common/constants/mu,ro,eo,pi,cang,ro2,dipol,emax,m1,m2,
9        ?xe,xk,xn,mconst,correct,romax,inatom,icoord,iic
10        common/charge/pcharge(len)
11        common/coordinates/fx(len),fy(len),fz(len),
12       ?ox(len),oy(len),oz(len)
13        common/ljparameters/eolj(len),rolj(len),eox4(len),
14       ?ro6lj(len),ro12lj(len),dro6(len),dro12(len)
15        common/trajectory/sw1,sw2,dtsf1,dtsf2,cmin,ifail,ifailc,inwr
16        common/angles/theta,phi,gamma
17        common/xrandom/i1,i2,i3,i4,i5,i6
18 C
19        rx=0.d0
20        ry=0.d0
21        rz=0.d0
22        e00=0.d0
23        de00x=0.d0
24        de00y=0.d0
25        de00z=0.d0
26        sum1=0.d0
27        sum2=0.d0
28        sum3=0.d0
29        sum4=0.d0
30        sum5=0.d0
31        sum6=0.d0
32        dmax=2.d0*romax
33 C
34        do 1100 iatom=1,inatom
35        xx=x-fx(iatom)
36        xx2=xx*xx
37        yy=y-fy(iatom)
38        yy2=yy*yy
39        zz=z-fz(iatom)
40        zz2=zz*zz
```

```
41        rxyz2=xx2+yy2+zz2
42        rxyz=dsqrt(rxyz2)
43        if(rxyz.lt.dmax) dmax=rxyz
44        rxyz3=rxyz2*rxyz
45        rxyz5=rxyz3*rxyz2
46        rxyz6=rxyz5*rxyz
47        rxyz8=rxyz5*rxyz3
48        rxyz12=rxyz6*rxyz6
49        rxyz14=rxyz12*rxyz2
50  c     LJ potential
51        e00=e00+(eox4(iatom)*((ro12lj(iatom)/rxyz12)-
52       ?(ro6lj(iatom)/rxyz6)))
53  c     LJ derivative
54        de00=eox4(iatom)*((dro6(iatom)/rxyz8)-
55       ?(dro12(iatom)/rxyz14))
56        de00x=de00x+(de00*xx)
57        de00y=de00y+(de00*yy)
58        de00z=de00z+(de00*zz)
59  c     ion-induced dipole potential
60        if(pcharge(iatom).eq.0.d0) goto 1100
61        rxyz3i=pcharge(iatom)/rxyz3
62        rxyz5i=-3.d0*pcharge(iatom)/rxyz5
63        rx=rx+(xx*rxyz3i)
64        ry=ry+(yy*rxyz3i)
65        rz=rz+(zz*rxyz3i)
66  c     ion-induced dipole derivative
67        sum1=sum1+(rxyz3i+(xx2*rxyz5i))
68        sum2=sum2+(xx*yy*rxyz5i)
69        sum3=sum3+(xx*zz*rxyz5i)
70        sum4=sum4+(rxyz3i+(yy2*rxyz5i))
71        sum5=sum5+(yy*zz*rxyz5i)
72        sum6=sum6+(rxyz3i+(zz2*rxyz5i))
73  c
74   1100 continue
75  c
76        pot=e00-(dipol*((rx*rx)+(ry*ry)+(rz*rz)))
77        dpotx=de00x-(dipol*((2.d0*rx*sum1)+(2.d0*ry*sum2)
78       ?+(2.d0*rz*sum3)))
79        dpoty=de00y-(dipol*((2.d0*rx*sum2)+(2.d0*ry*sum4)
80       ?+(2.d0*rz*sum5)))
81        dpotz=de00z-(dipol*((2.d0*rx*sum3)+(2.d0*ry*sum5)
82       ?+(2.d0*rz*sum6)))
83  c
84        return
85        end
```

By analysing the Mobcal *dljpot* function, one can notice that the function is used in four different ways. In HPCCS it was splitted in four overloaded functions where the arguments are different but all calculate the same potential and return the necessary values.

In Listing 4.9 only three constants are used, the x, y and z molecular coordinates, used before the angle calculation. A careful analysis of this function reveal that it can be easily vectorized.

Listing 4.9: He Pontential Calculation on HPCCS

```
1
2  double dljpotHe(const double x,const double y,const double z){
3
4  //  Subroutine to calculate L-J + ion-dipole potential.
5      double rxyz[7] __attribute__((aligned(16)));
6      double rx, ry, rz, e00, pot;
7      double xx, xx2, yy, yy2, zz, zz2, rxyz3i;
8
9      rx = 0.0;
10     ry = 0.0;
```

```
    rz = 0.0;
    e00 = 0.0;
    #if defined(__INTEL_COMPILER)
    #pragma vector aligned
    #endif
    #pragma omp simd
    for (unsigned int iatom = 0; iatom < numberOfAtoms; iatom++){
        xx = x - molecule->fx[iatom];
        xx2 = xx * xx;
        yy = y - molecule->fy[iatom];
        yy2 = yy * yy;
        zz = z - molecule->fz[iatom];
        zz2 = zz * zz;
        rxyz[0] = xx2 + yy2 + zz2;
        rxyz[1] = sqrt(rxyz[0]);
        rxyz[2] = rxyz[0] * rxyz[1];
        rxyz[3] = rxyz[2] * rxyz[0];
        rxyz[4] = rxyz[3] * rxyz[1];
        rxyz[5] = rxyz[3] * rxyz[2];
        rxyz[6] = rxyz[4] * rxyz[4];

        // LJ potential
        e00 += (molecule->eox4[iatom] * ((molecule->ro12lj[iatom] /
            rxyz[6]) - (molecule->ro6lj[iatom] / rxyz[4])));

        // ion-induced dipole potential
        rxyz3i = molecule->charge[iatom] / rxyz[2];
        rx += xx * rxyz3i;
        ry += yy * rxyz3i;
        rz += zz * rxyz3i;
    }
    pot =  e00 - (dipol * ((rx * rx) + (ry * ry) + (rz * rz)));

    return pot;
}
```

Listing 4.10: He Pontential Calculation on HPCCS

```
double dljpotHe(const double x,const double y,const double z,
    double *fx, double *fy, double *fz){

//  Subroutine to calculate L-J + ion-dipole potential.
    double rxyz[7] __attribute__((aligned(16)));
    double rx,ry,rz;
    double xx,xx2,yy, yy2, zz,zz2,rxyz3i;
    double pot, e00 = 0.0;;
    rx = 0.0;
    ry = 0.0;
    rz = 0.0;

    #if defined(__INTEL_COMPILER)
    #pragma vector aligned
    #endif
    #pragma omp simd
    for (unsigned int iatom = 0; iatom < numberOfAtoms; ++iatom){
        xx = x - fx[iatom];
        xx2 = xx * xx;
        yy = y - fy[iatom];
        yy2 = yy * yy;
        zz = z - fz[iatom];
        zz2 = zz * zz;
        rxyz[0] = xx2 + yy2 + zz2;
        rxyz[1] = sqrt(rxyz[0]);
        rxyz[2] = rxyz[0] * rxyz[1];
        rxyz[3] = rxyz[2] * rxyz[0];
        rxyz[4] = rxyz[3] * rxyz[1];
        rxyz[5] = rxyz[3] * rxyz[2];
        rxyz[6] = rxyz[4] * rxyz[4];

```

```
31          // LJ potential
32          e00 += (molecule->eox4[iatom] * ((molecule->ro12lj[iatom] /
                rxyz[6]) - (molecule->ro6lj[iatom] / rxyz[4])));
33
34      // ion-induced dipole potential
35      rxyz3i = molecule->charge[iatom] / rxyz[2];
36      rx += xx * rxyz3i;
37      ry += yy * rxyz3i;
38      rz += zz * rxyz3i;
39
40      }
41      pot =  e00 - (dipol * ((rx * rx) + (ry * ry) + (rz * rz)));
42      return pot;
43  }
```

To achieve auto vectorization at Listing 4.11 first all arrays must be memory aligned, which means the data must be consecutive with no gaps between them. Starting on GCC 7.2 the operator *new* will do it automatically by using the $\_\_attribute\_\_((aligned(16)))$ which is responsible to align exactly n bytes (in this example 16 bytes). This will help the compiler to vectorize the loop using the SSE/AVX extensions.

In Listing 4.8 at line 60 a change in program flow which percludes the programmer to vectorize the loop. In Listing 4.11 and Listing 4.12 this was handled creating by first creating an array *rxyz_vec* to store all values inside the loop. A second loop was created outside the first loop to check the *rxyz_vec* values as seen in Listing 4.11 when calculating the *dmaxx* value. After re-structuring the code this way the loop was vectorized and performance was improved.

Listing 4.11: He Pontential Calculation on HPCCS

```
1  void dljpotHe(const double x,const double y,const double z, double
       *pot, double *dmax, double *dpotx, double *dpoty, double *dpotz,
        double *fx, double *fy, double *fz){
2
3  //  Subroutine to calculate L-J + ion-dipole potential.
4
5       double rxyz[8] __attribute__((aligned(16)));
6       double  de00, rxyz3i, rxyz5i;
7       double xx, xx2, yy, yy2, zz, zz2;
8       double e00, de00x, de00y, de00z, dmaxx, eox4;
9       double sum1, sum2, sum3, sum4, sum5, sum6;
10      double *rxyz_vec __attribute__((aligned(16)));
11
12      double rx = 0.0, ry = 0.0, rz = 0.0;
13      e00 = 0.0;
14      de00x = 0.0;
15      de00y = 0.0;
16      de00z = 0.0;
17      sum1 = 0.0;
18      sum2 = 0.0;
19      sum3 = 0.0;
20      sum4 = 0.0;
21      sum5 = 0.0;
22      sum6 = 0.0;
23      dmaxx = 2.0 * romax;
24      rxyz_vec = new double[numberOfAtoms];
25
26      #if defined(__INTEL_COMPILER)
27      #pragma vector aligned
28      #endif
29      #pragma omp simd
30      for (unsigned int iatom = 0; iatom < numberOfAtoms; ++iatom){
31
32          xx = x - fx[iatom];
33          xx2 = xx * xx;
```

```
34          yy = y - fy[iatom];
35          yy2 = yy * yy;
36          zz = z - fz[iatom];
37          zz2 = zz * zz;
38          rxyz[0] = xx2 + yy2 + zz2;
39          rxyz[1] = sqrt(rxyz[0]);
40          rxyz_vec[iatom] = rxyz[1];
41          rxyz[2] = rxyz[0] * rxyz[1];
42          rxyz[3] = rxyz[2] * rxyz[0];
43          rxyz[4] = rxyz[3] * rxyz[1];
44          rxyz[5] = rxyz[3] * rxyz[2];
45          rxyz[6] = rxyz[4] * rxyz[4];
46          rxyz[7] = rxyz[6] * rxyz[0];
47          eox4 = molecule->eox4[iatom];
48          // LJ potential
49          e00 += (eox4 * ((molecule->ro12lj[iatom] / rxyz[6]) - (
               molecule->ro6lj[iatom] / rxyz[4])));
50
51          // LJ derivative
52          de00 = eox4 * ((molecule->dro6[iatom] / rxyz[5]) - (
               molecule->dro12[iatom] / rxyz[7]));
53          de00x += de00 * xx;
54          de00y += de00 * yy;
55          de00z += de00 * zz;
56
57          // ion-induced dipole potential
58          rxyz3i = molecule->charge[iatom] / rxyz[2];
59          rxyz5i = -3.0 * molecule->charge[iatom] / rxyz[3];
60          rx += xx * rxyz3i;
61          ry += yy * rxyz3i;
62          rz += zz * rxyz3i;
63          //  ion-induced dipole derivative
64          sum1 += rxyz3i + (xx2 * rxyz5i);
65          sum2 += xx * yy * rxyz5i;
66          sum3 += xx * zz * rxyz5i;
67          sum4 += rxyz3i + (yy2 * rxyz5i);
68          sum5 += yy * zz * rxyz5i;
69          sum6 += rxyz3i + (zz2 * rxyz5i);
70  }
71      #if defined(__INTEL_COMPILER)
72      #pragma vector aligned
73      #endif
74      for (unsigned int iatom = 0; iatom < numberOfAtoms; ++iatom){
75          if(rxyz_vec[iatom] < dmaxx)  dmaxx = rxyz_vec[iatom];
76      }
77      *dmax = dmaxx;
78      delete[] rxyz_vec;
79      *pot = e00 - (dipol * ((rx * rx) + (ry * ry) + (rz * rz)));
80      *dpotx = de00x -(dipol * ((2.0 * rx * sum1) + (2.0 * ry * sum2)
            + (2.0 * rz * sum3)));
81      *dpoty = de00y -(dipol * ((2.0 * rx * sum2) + (2.0 * ry * sum4)
            + (2.0 * rz * sum5)));
82      *dpotz = de00z -(dipol * ((2.0 * rx * sum3) + (2.0 * ry * sum5)
            + (2.0 * rz * sum6)));
83  }
```

Listing 4.12: He Pontential Calculation on HPCCS

```
1  void dljpotHe(const double x,const double y,const double z, double
       *pot, double *dmax, double *dpotx, double *dpoty, double *dpotz)
       {
2
3  //  Subroutine to calculate L-J + ion-dipole potential.
4
5      double rxyz[8] __attribute__((aligned(16)));
6      double rx, ry, rz, e00, de00, de00x, de00y, de00z;
7      double sum1, sum2, sum3, sum4, sum5, sum6, dmaxx, eox4;
8      double xx, xx2, yy, yy2, zz, zz2, rxyz3i, rxyz5i;
```

```
9       double *rxyz_vec __attribute__((aligned(16)));
10
11      rx = 0.0;
12      ry = 0.0;
13      rz = 0.0;
14      e00 = 0.0;
15      de00x = 0.0;
16      de00y = 0.0;
17      de00z = 0.0;
18      sum1 = 0.0;
19      sum2 = 0.0;
20      sum3 = 0.0;
21      sum4 = 0.0;
22      sum5 = 0.0;
23      sum6 = 0.0;
24      dmaxx = 2.0 * romax;
25      rxyz_vec = new double[numberOfAtoms];
26      #if defined(__INTEL_COMPILER)
27      #pragma vector aligned
28      #endif
29      #pragma omp simd
30      for (unsigned int iatom = 0; iatom < numberOfAtoms; iatom++){
31          xx = x - molecule->fx[iatom];
32          xx2 = xx * xx;
33          yy = y - molecule->fy[iatom];
34          yy2 = yy * yy;
35          zz = z - molecule->fz[iatom];
36          zz2 = zz * zz;
37          rxyz[0] = xx2 + yy2 + zz2;
38          rxyz[1] = sqrt(rxyz[0]);
39          rxyz_vec[iatom] = rxyz[1];
40          rxyz[2] = rxyz[0] * rxyz[1];
41          rxyz[3] = rxyz[2] * rxyz[0];
42          rxyz[4] = rxyz[3] * rxyz[1];
43          rxyz[5] = rxyz[3] * rxyz[2];
44          rxyz[6] = rxyz[4] * rxyz[4];
45          rxyz[7] = rxyz[6] * rxyz[0];
46
47          // LJ potential
48          eox4 = molecule->eox4[iatom];
49          e00 += (eox4 * ((molecule->ro12lj[iatom] / rxyz[6]) - (
                molecule->ro6lj[iatom] / rxyz[4])));
50
51          // LJ derivative
52          de00 = eox4 * ((molecule->dro6[iatom] / rxyz[5]) - (
                molecule->dro12[iatom] / rxyz[7]));
53          de00x += de00 * xx;
54          de00y += de00 * yy;
55          de00z += de00 * zz;
56
57          // ion-induced dipole potential
58          rxyz3i = molecule->charge[iatom] / rxyz[2];
59          rxyz5i = -3.0 * molecule->charge[iatom] / rxyz[3];
60          rx += xx * rxyz3i;
61          ry += yy * rxyz3i;
62          rz += zz * rxyz3i;
63          //  ion-induced dipole derivative
64          sum1 += rxyz3i + (xx2 * rxyz5i);
65          sum2 += xx * yy * rxyz5i;
66          sum3 += xx * zz * rxyz5i;
67          sum4 += rxyz3i + (yy2 * rxyz5i);
68          sum5 += yy * zz * rxyz5i;
69          sum6 += rxyz3i + (zz2 * rxyz5i);
70
71
72  }
73      #if defined(__INTEL_COMPILER)
74      #pragma vector aligned
75      #endif
76      for (unsigned int iatom = 0; iatom < numberOfAtoms; ++iatom){
```

```
77              if(rxyz_vec[iatom] < dmaxx)  dmaxx = rxyz_vec[iatom];
78      }
79      *dmax = dmaxx;
80      delete[] rxyz_vec;
81      *pot = e00 - (dipol * ((rx * rx) + (ry * ry) + (rz * rz)));
82      *dpotx = de00x -(dipol * ((2.0 * rx * sum1) + (2.0 * ry * sum2)
            + (2.0 * rz * sum3)));
83      *dpoty = de00y -(dipol * ((2.0 * rx * sum2) + (2.0 * ry * sum4)
            + (2.0 * rz * sum5)));
84      *dpotz = de00z -(dipol * ((2.0 * rx * sum3) + (2.0 * ry * sum5)
            + (2.0 * rz * sum6)));
85  }
```

The next section will present the strategies using OpenMP and MPI + OpenMP to parallelize the HPCCS and improve its performance.

4.7 CCS Calculation (OpenMP)

The *gsang* function must be called for CCS calculation. Listing 4.13 shows how the function is called, at line 27, where it uses a triple loop to calculate the velocity v, and the parameter b. The parameter *erat* is not necessary since it is used only inside each trajectory for energy conservation, the *ang* parameter is the angle calculated which is returned by the function, *d1* is the value of position y and the *istep* parameter is the current step to be calculated.

The triple loop will iterate over *inp*, *imp* and *itn* representing the Monte Carlo sampling to calculate the final values after each calculated angle by the *gsang* function. The molecule is rotated at each iteration using random values for each position (x, y, z). They are generated using a pseudorandom number generators called Ranlux, commonly used in computational physics and chemistry [10].

Mobcal was developed without caring about the variable scope. There are many variables sharing their values in many code parts. This does not cause any problem in a serial execution. If the code is parallelized as it is, the threads will change the variable values causing wrong angle values.

Since each angle calculation is independent, to have it running in parallel, the code must be rewritten having the variables private to each thread.

Listing 4.13: Gsang Function on Mobcal

```
1
2       do 4040 ic=1,itn
3         if(ip.eq.1) write(*,681) ic
4     681 format(/1x,'cycle number, ic =',i3)
5         om11st(ic)=0.d0
6         om12st(ic)=0.d0
7         om13st(ic)=0.d0
8         om22st(ic)=0.d0
9         do 4010 ig=1,inp
10        gst2=pgst(ig)*pgst(ig)
11        v=dsqrt((gst2*eo)/(0.5d0*mu))
12        if(ip.eq.1) write(*,682) ic,ig,gst2,v
13    682 format(/1x,'ic =',i3,1x,'ig =',i4,1x,'gst2 =',1pe11.4,
14       ?1x,'v =',e11.4)
15        temp1=0.d0
16        temp2=0.d0
17  c
18        do 4000 im=1,imp
```

```
19        if(ip.eq.1.and.im.eq.1) write(*,683)
20    683 format(/5x,'b/A',8x,'ang',6x,'(1-cosX)',4x,'e ratio',4x,'
       theta',
21       ?7x,'phi',7x,'gamma')
22        rnb=xrand()
23        call rantate
24        bst2=rnb*b2max(ig)
25        b=ro*dsqrt(bst2)
26  c
27        call gsang(v,b,erat,ang,d1,istep)
28        hold1=1.d0-dcos(ang)
29        hold2=dsin(ang)*dsin(ang)
30        if(ip.eq.1) write(*,684) b*1.d10,ang*cang,hold1,erat,
31       ?theta*cang,phi*cang,gamma*cang
32    684 format(1x,1pe11.4,7(e11.4))
33        temp1=temp1+(hold1*b2max(ig)/dfloat(imp))
34        temp2=temp2+(1.5d0*hold2*b2max(ig)/dfloat(imp))
35   4000 continue
```

By carefully analysing the angle calculation on Mobcal, one can notice that the trajectories are independent of each other and thus can be calculated in parallel. Thus HPCCS runs the angle calculation in parallel, using all available resources from a single cluster node or desktop using a multicore processor and vector units.

To run the code in parallel the values of *v, b* and random values for *rotate* function are stored into vectors *v_vec, b_vec, rng2, rng3, rng4*. All vectors have the same size and they are aligned in memory. Their values are calculated and stored before the parallel execution.

The Ranlux random generator was replaced on HPCCS by Mersenne Twister [19], the most widely used general-purpose PRNG [18].

To run HPCCS in parallel some techniques were applied as follows.

- Variables must not be shared by any other function, the variables will be kept inside the *gsang*, *rotate*, *diffeq_deriv* and *dljpotHe* functions.
- Each thread will execute the entire angle calculation independently. Since the execution is totally serial, it must be optimized to use the processor resources to process it faster.

Once the above code transformations are performed, the program can be parallelized to run on multicore processors on a single shared memory computer node. To split the calculation on all available cores it was used OpenMP [4], which is a well-known API available for C/C++ and Fortran languages, used to parallelize code blocks using shared memory model. It consists in using directives *#pragmas* above the loops or region splitted as tasks that run in parallel. The Listing 4.14 shows how OpenMP was used in the design of HPCCS.

- At line **1** the OpenMP directive starts with *#pragma omp*.
- The *parallel* clause will start the available threads to execute the parallel region.
- The *for* is responsible to parallelize the for loop bellow it.

- The *private* clause will consider the variables in this case (*ang*, *id*) private to each thread execution. A copy of them is made to not share their value avoiding wrong values to be written or read at each iteration.

- OpenMP has some clauses to schedule the threads to execute the loop, it does by splitting the loop iterations according to the *schedule* word. In this case, the schedule clause used the ***dynamic*** parameter, which divides the loop iterations into chunks. Each thread executes a chunk of iterations and then requests another chunk, until there are no more chunks available. There is no particular order in which the chunks are distributed to the threads. The order changes each time when we execute the for loop.

- The time spent to each angle be calculated in *gsang* differs from each thread. The calculations are be executed one after another without any synchronization, having all threads busy most of the execution time while in the parallel region.

- The *omp for* has an implicit barrier. The code after the loop will only be executed when all threads finish their work.

Listing 4.14: Parallelized region for angle calculation using OpenMP

```
1         #pragma omp parallel for private(id,ang) schedule(dynamic)
2         for (id = 0; id < total; ++id){
3             ang = gsangHe(v_vec[id],b_vec[id],rng2[id],rng3[id],
                 rng4[id],numberOfAtoms);
4             temp_vec[id] = ((1.0 - cos(ang)) * b2max_vec[id] /
                 float(imp));
5             temp1_vec[id] = (1.5 * (sin(ang) * sin(ang)) *
                 b2max_vec[id] / float(imp));
6         }
```

The *gsang* function is responsible for calling the other functions, returning at the end the angles to compute the CCS. The *rotate* function, as described before, rotate randomly the molecule for each execution. The *diffeq_ deriv* is the integration subroutine, which uses 5th order Runge-Kutta-Gill to initiate and 5th order Adams-Moulton predictor-corrector to propagate.

After each angle calculation, they are stored into two arrays for CCS calculation. According to Listing 4.15, from line 2 to 24 it ends the Monte Carlo integration, having the arrays calculated using the angles values. As described before, the Monte Carlo samples are calculated using three loops,In Listing 4.5 loop collapsing in performed in order to accelerate the calculation. The CCS value is stored into the *cs* variable. The other variables are as follows:

- ***mob*** is the Average (second order) TM mobility.

- ***1.0/mob*** is the Inverse average (second order) TM mobility.

- ***cs*** is the Average TM cross section **(CCS)**.

- ***sdevpc*** is the Standard deviation.

- ***itn * inp * imp*** is the Total number of Trajectories.
- ***trajlost*** is the Number of lost trajectories.

Listing 4.15: Code for CCS calculation after all angles are calculated

```
1
2     for(int i = 0; i < total / imp; i++){
3           double count = 0, count1 = 0;
4           for (int j = 0; j < imp; j++){
5              count += temp_vec[i * imp + j];
6              count1 += temp1_vec[i * imp + j];
7           }
8           temp_vec2[i] = count;
9           temp1_vec2[i] = count1;
10    }
11
12    id= 0;
13
14    for (ic = 0; ic < itn; ++ic){
15        for (ig = 0; ig < inp; ++ig){
16            om11st[ic] += temp_vec2[id] * wgst[ig];
17            om12st[ic] += temp_vec2[id] * pgst[ig] * pgst[ig] *
                 wgst[ig] * (1.0 / (3.0 * tst));
18            om13st[ic] += temp_vec2[id] * (pow(pgst[ig],4)) * wgst[
                 ig] * (1.0 / (12.0 * tst * tst));
19            om22st[ic] += temp1_vec2[id] * pgst[ig] * pgst[ig] *
                 wgst[ig] * (1.0 / (3.0 * tst));
20            q1st[ig] += temp_vec2[id];
21            q2st[ig] += temp1_vec2[id];
22            id++;
23        }
24    }
25 /* End of Monte Carlo */
26
27    if(ifailc < ifail){
28    //Calculate running averages
29
30    hold1 = 0.0;
31    hold2 = 0.0;
32    for (ic = 0; ic < itn; ++ic){
33        temp = 1.0 / (mconst / (sqrt(temperature) * om11st[ic] * pi
              * ro * ro));
34        hold1 += om11st[ic];
35        hold2 += temp;
36    }
37
38    mom11st = 0.0;
39    mom12st = 0.0;
40    mom13st = 0.0;
41    mom22st = 0.0;
42    for (ic = 0; ic < itn; ++ic){
43        mom11st += om11st[ic];
44        mom12st += om12st[ic];
45        mom13st += om13st[ic];
46        mom22st += om22st[ic];
47    }
48
49    mom11st /= float(itn);
50    mom12st /= float(itn);
51    mom13st /= float(itn);
52    mom22st /= float(itn);
53    sdom11st = 0.0;
54
55    for (ic = 0; ic < itn; ++ic){
56        hold = mom11st - om11st[ic];
57        sdom11st += (hold * hold);
```

```
58      }
59
60      sdom11st = sqrt(sdom11st / float(itn));
61      cs = mom11st * pi * ro * ro;
62      sdevpc = (100.0 * sdom11st)/mom11st;
63
64      //Use omegas to obtain higher order correction factor to
           mobility
65       ayst = mom22st / mom11st;
66       best = ((5.0 * mom12st) - (4.0 * mom13st)) / mom11st;
67       cest = mom12st/mom11st;
68       term = ((4.0 * ayst) / (15.0)) + (0.5 * (pow((m2-m1),2)) / (m1
             * m2));
69       u2 = term - (0.08333 * (2.4 * best + 1.0) * (m1 / m2));
70       w = (m1 / m2);
71       delta = ((pow(((6.0 * cest)-5.0),2)) * w) / (60.0 * (1.0 + u2)
            );
72       f = 1.0 / (1.0 - delta);
73       mob = (mconst * f)/(sqrt(temperature) * cs);
74      }
```

4.8 CCS Calculation (MPI + OpenMP)

HPCCS was also designed to use MPI [6], which is a standard for message passing on a distribute memory system. It distributes the work across the multiples cluster's nodes having a program copy on each node, which process a slice of the input data each.

MPI performs collective communications between the processes, and recieve data across the various cluster nodes. Sending all data to all nodes is called *broadcast*. If the program data are splitted to send only the necessary amount of data to the nodes is a *scatter*, MPI performs a scatter operation, thus avoiding unnecessary data traffic across the cluster. The processes wait until all members of the processes group of the same communicator, have reached the synchronization point.

After the data is processed (thru a reduction operation) one process will collect the data from the other processes, doing the final computation. There are two ways to receive the data, one is the *reduction*, in which the processes reduce a large dataset to a scalar like a min or max. The other is *gather*, which receives the the data from other process and put them together into an array. Figure 4.5 illustrates the collective communications.

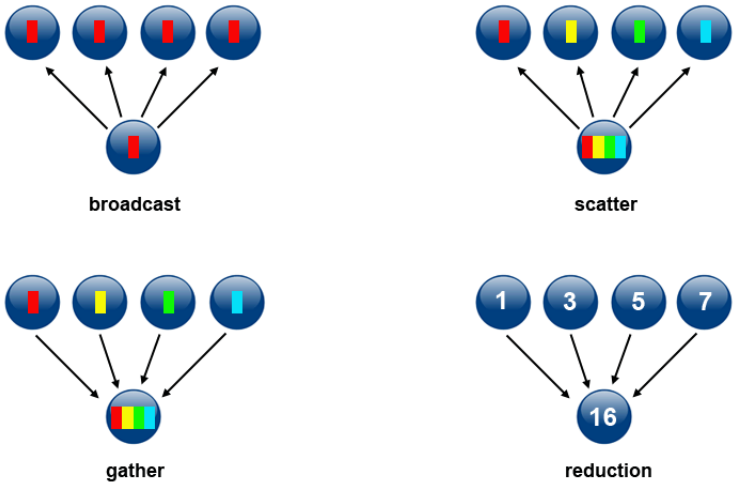

Figure 4.5: MPI Collective Communication (Blaise Barney, LLNL)

HPCCS MPI + OpenMP version uses one process per cluster node. The master process (rank 0) sends the data using scatter communication, reducing the data traffic and communication between processes. Using this hybrid approach, each process will trigger all available threads on each node to compute the angles as it does in the single node OpenMP version but using only an array slice.

Listing 4.16 shows the use of *broadcast* to send the common values to all process and *scatter* to send the arrays slice to the *gsang* for function computation. After all angles are computed, a *gather* is used to get all angles and store then into the *ang_vec* array. The *rank 0* computes the final values using the angles previously calculated, to store them into *temp_vec* and *temp1_vec array*. The *rank 0* executes the serial calculations and only the final result is calculated using OpenMP.

Listing 4.16: Code for CCS calculation after all angles are calculated

```
1
2       //BroadCast
3       MPI_Bcast(&dipol, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
4       MPI_Bcast(&m1, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
5       MPI_Bcast(&mconst, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
6       MPI_Bcast(&mu, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
7       MPI_Bcast(&m2, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
8       MPI_Bcast(&romax, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
9       MPI_Bcast(&theta, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
10      MPI_Bcast(&phi, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
11      MPI_Bcast(&agamma, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
12      MPI_Bcast(&ifailc, 1, MPI_INT, 0, MPI_COMM_WORLD);
13      MPI_Bcast(&temperature, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
14
15      //Broadcast Molecule Attributes
16      MPI_Bcast(moleculeFx, numberOfAtoms, MPI_DOUBLE, 0,
           MPI_COMM_WORLD);
17      MPI_Bcast(moleculeFy, numberOfAtoms, MPI_DOUBLE, 0,
           MPI_COMM_WORLD);
18      MPI_Bcast(moleculeFz, numberOfAtoms, MPI_DOUBLE, 0,
           MPI_COMM_WORLD);
19      MPI_Bcast(moleculeOx, numberOfAtoms, MPI_DOUBLE, 0,
           MPI_COMM_WORLD);
20      MPI_Bcast(moleculeOy, numberOfAtoms, MPI_DOUBLE, 0,
           MPI_COMM_WORLD);
```

```
21      MPI_Bcast(moleculeOz, numberOfAtoms, MPI_DOUBLE, 0,
            MPI_COMM_WORLD);
22      MPI_Bcast(moleculeCharge, numberOfAtoms, MPI_DOUBLE, 0,
            MPI_COMM_WORLD);
23      MPI_Bcast(moleculeXmass, numberOfAtoms, MPI_DOUBLE, 0,
            MPI_COMM_WORLD);
24      MPI_Bcast(moleculeEox4, numberOfAtoms, MPI_DOUBLE, 0,
            MPI_COMM_WORLD);
25      MPI_Bcast(moleculeRo6lj, numberOfAtoms, MPI_DOUBLE, 0,
            MPI_COMM_WORLD);
26      MPI_Bcast(moleculeRo12lj, numberOfAtoms, MPI_DOUBLE, 0,
            MPI_COMM_WORLD);
27      MPI_Bcast(moleculeDro6, numberOfAtoms, MPI_DOUBLE, 0,
            MPI_COMM_WORLD);
28      MPI_Bcast(moleculeDro12, numberOfAtoms, MPI_DOUBLE, 0,
            MPI_COMM_WORLD);
29
30      //MPIScatter
31      MPI_Scatter(v_vec, totalSimulationsByProcs, MPI_DOUBLE,
            v_vec_partial, totalSimulationsByProcs, MPI_DOUBLE, 0,
            MPI_COMM_WORLD);
32      MPI_Scatter(b_vec, totalSimulationsByProcs, MPI_DOUBLE,
            b_vec_partial, totalSimulationsByProcs, MPI_DOUBLE, 0,
            MPI_COMM_WORLD);
33      MPI_Scatter(rng2, totalSimulationsByProcs, MPI_DOUBLE,
            rng2_partial,totalSimulationsByProcs, MPI_DOUBLE, 0,
            MPI_COMM_WORLD);
34      MPI_Scatter(rng3, totalSimulationsByProcs, MPI_DOUBLE,
            rng3_partial, totalSimulationsByProcs, MPI_DOUBLE, 0,
            MPI_COMM_WORLD);
35      MPI_Scatter(rng4, totalSimulationsByProcs, MPI_DOUBLE,
            rng4_partial, totalSimulationsByProcs, MPI_DOUBLE, 0,
            MPI_COMM_WORLD);
36
37    parallelGsang(ang_vec_partial, v_vec_partial, b_vec_partial,
          rng2_partial,
38       rng3_partial, rng4_partial, totalSimulationsByProcs,
            numberOfAtoms);
39
40    MPI_Gather(ang_vec_partial, totalSimulationsByProcs, MPI_DOUBLE,
          ang_vec, totalSimulationsByProcs, MPI_DOUBLE, 0,
         MPI_COMM_WORLD);
41
42    //Final Result
43     if (my_rank == 0){
44
45          #pragma omp parallel for private(id)
46          for (id = 0; id < totalSimulations; ++id){
47              temp_vec[id] = ((1.0 - cos(ang_vec[id])) * b2max_vec[id
                    ] / float(imp));
48              temp1_vec[id] = (1.5 * (sin(ang_vec[id]) * sin(ang_vec[
                    id])) * b2max_vec[id] / float(imp));
49          }
```

Chapter 5

Experimental Results

The experimental results were calculated using the Center for Computing in Engineering & Sciences cluster (Kahuna) at the University of Campinas.

Two types of experiments were performed. The first using only one node running OpenMP version using one single node and the second using the hybrid version MPI + OpenMP using multiples nodes. They were executed using the nodes with the following configuration: Intel Xeon E5-2670 v3 with 2.30GHz and 48 threads. The turbo boost was disabled to have deterministic processor frequency, without any change during the experiments [15].

Ten executions were made for each experiment. The goal was to measure the time spent, the speedup and efficiency. Two groups of molecules were chosen to be simulated. The first group of molecules from Jurneczko and Barran [14], ranges from 432 to 4392 atoms in total. The other group, with protein complexes from Bush [2], contains 1674 to 32774 atoms.

In our test groups, we used two different buffer gases, N_2 and He. N_2 is more polarizable, thus allowing strong interactions with molecular ions. All experiments used only Helium gas since it produces simulations ≈ 5 times faster than Nitrogen, but calculations with Nitrogen were carried out to compare results with experimental data.

All experiments were compared with the original sequential non-optimized Mobcal version which was used as the speedup baseline. Unfortunately, when the optimization flag is turned on Mobcal does not calculate CCS.

HPCCS was compiled for serial and parallel execution using the following flags from GCC version 7.2 for both OpenMP and MPI + OpenMP version:

- -O3 -std=c++0x -fopenmp -mtune=native -march=native -mfma -ffast-math

The next section charts show the experimental results, which measure the time and speedup where comparing Mobcal to HPCCS sequential optimized version. The time spent in IMPACT is not compared to HPCCS, since IMPACT uses the PA method which is much faster and inaccurate than the trajectory method.

5.1 Results

This section presents the results, the first group is the speedup and time for each protein from Perdita's group. The second group is the protein complexes, where the number of atoms are greater than the first group, and their shapes are more complex.

The time spent on the second group is greater, but can be calculated in a feasible time using HPCCS, what was impossible by using the TM approach. The third group is the Bush group using MPI + OpenMP, which considerable reduces the execution time at the expenses of using much more computing resources. The experiments using Nitrogen were done only to check the results quality not the time and speedup since it takes longer to process.

For each simulation there are two charts, the first one shows the time spent on simulation and the second one is the speedup. In this graph the solid blue line represents the performance of parallelized the HPCSS with respect to its serial version.

5.2 Perdita's group

Below we present the results for all individual proteins from Perdita's group.

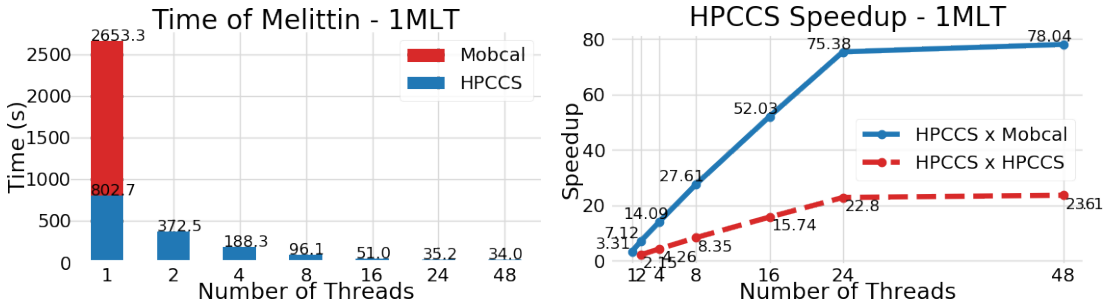

Figure 5.1: Time and Speedup of Pyruvate Kinase - 1AQF with 432 atoms.

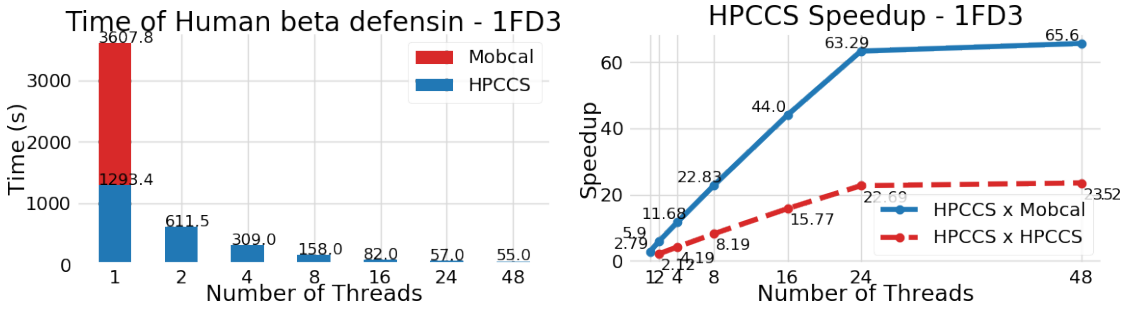

Figure 5.2: Time and Speedup of Human beta defensin - 1FD3 with 607 atoms.

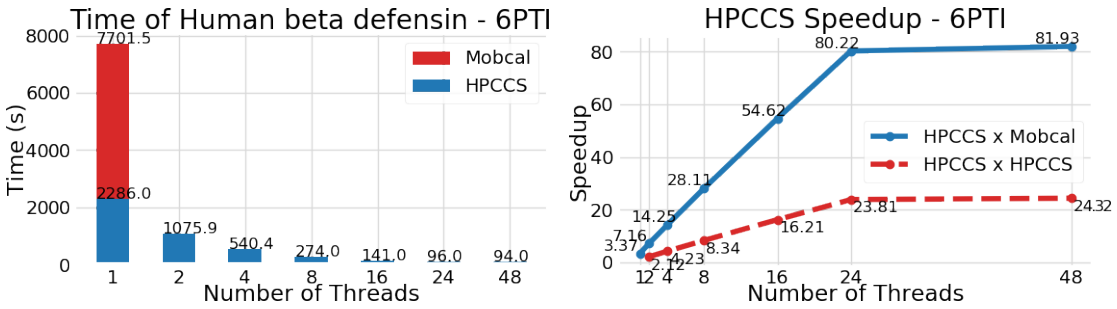

Figure 5.3: Time and Speedup of Bovine pancreatic trypsin inhibitor. - 6PTI with 880 atoms.

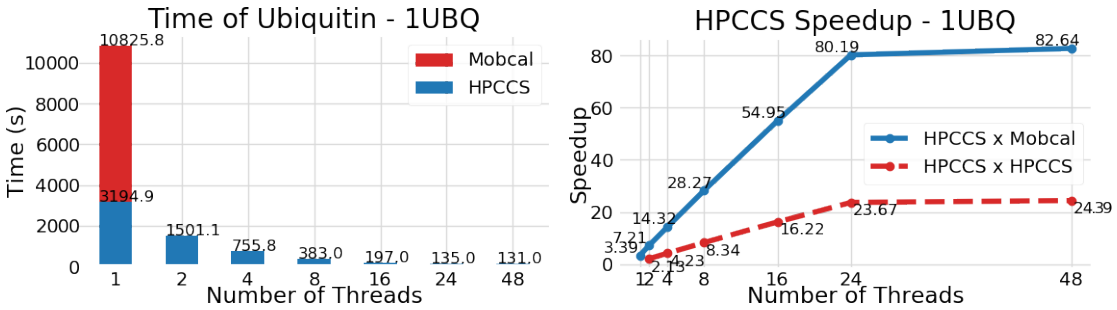

Figure 5.4: Time and Speedup of Ubiquitin - 1UBQ with 1235 atoms.

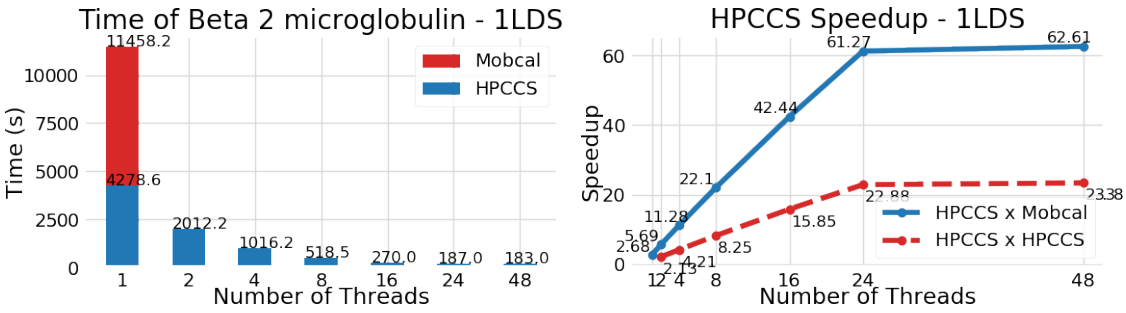

Figure 5.5: Time and Speedup of Beta 2 microglobulin - 1LDS with 1595 atoms.

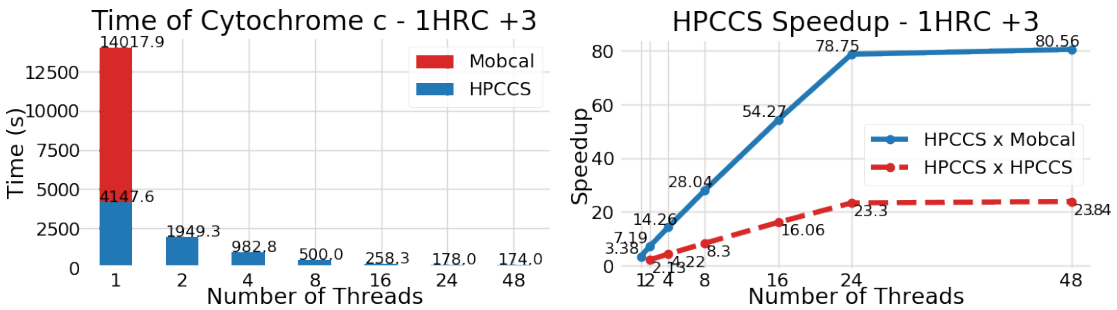

Figure 5.6: Time and Speedup of Cytochrome c - 1HRC +3 with 1666 atoms.

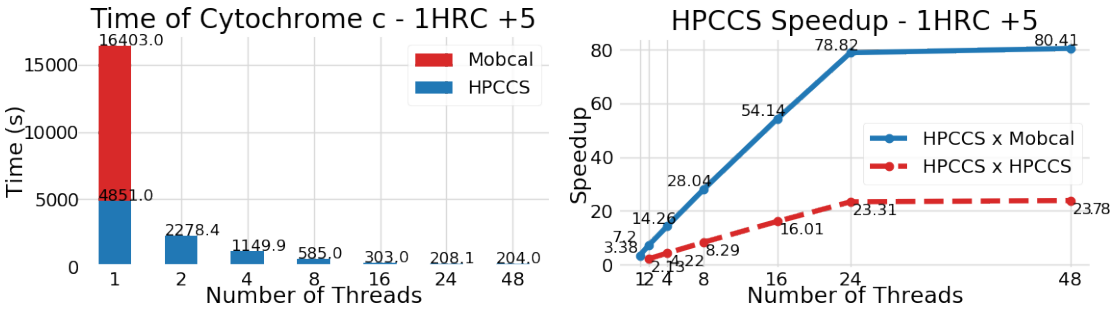

Figure 5.7: Time and Speedup of Cytochrome c - 1HRC +5 with 1668 atoms.

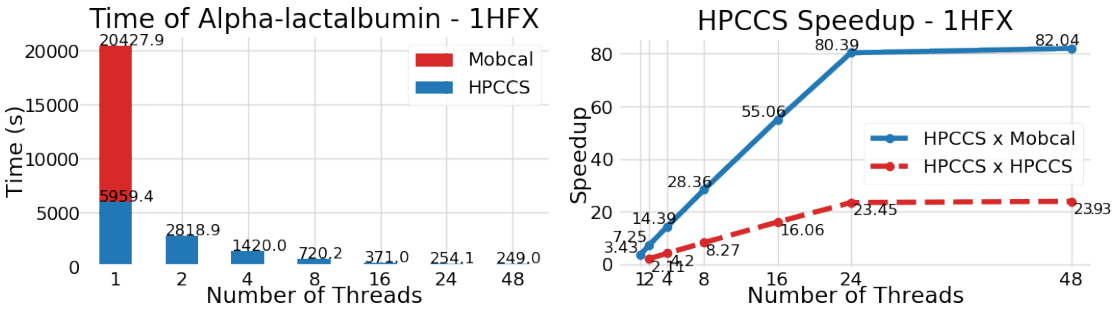

Figure 5.8: Time and Speedup of Alpha-lactalbumin - 1HFX with 1970 atoms.

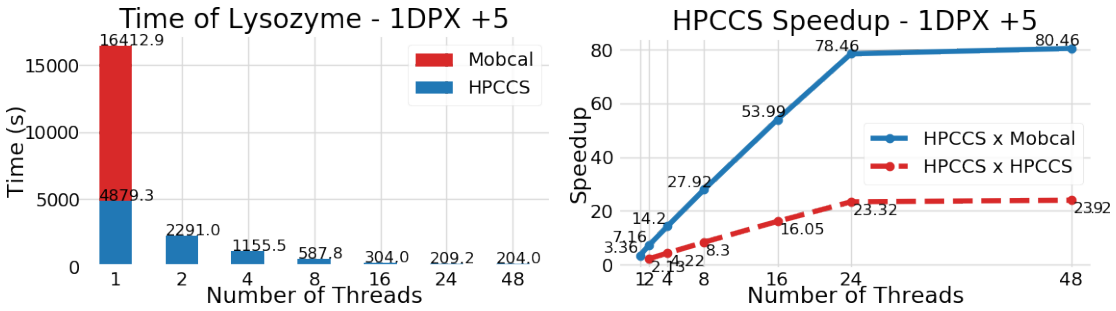

Figure 5.9: Time and Speedup of Lysozyme - 1DPX +5 with 1957 atoms.

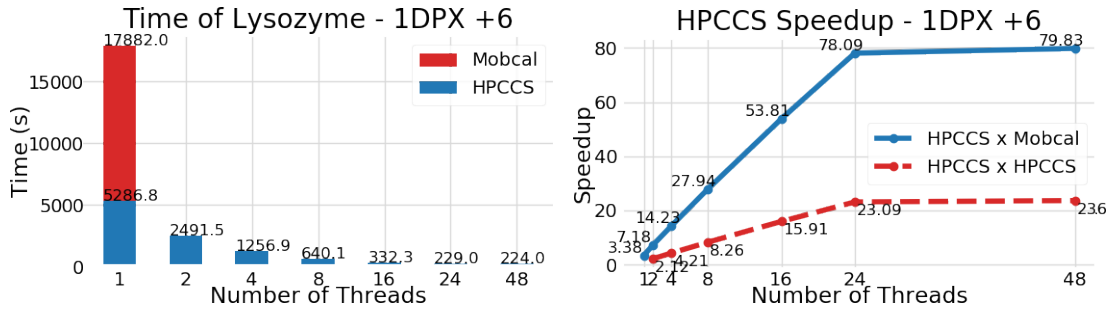

Figure 5.10: Time and Speedup of Lysozyme - 1DPX +6 with 1958 atoms.

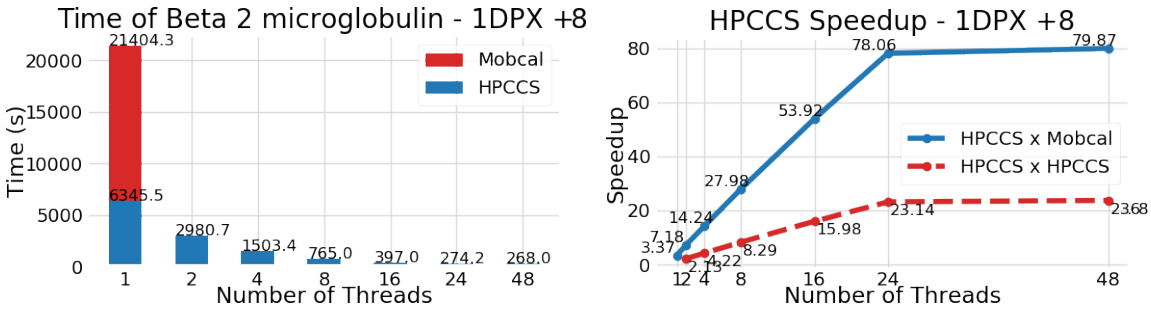

Figure 5.11: Time and Speedup of Lysozyme - 1DPX +8 with 1960 atoms.

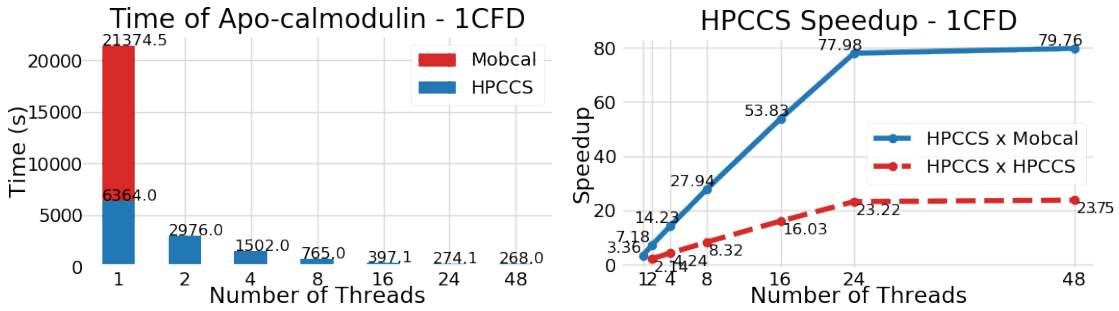

Figure 5.12: Time and Speedup of Apo-calmodulin - 1CFD with 2293 atoms.

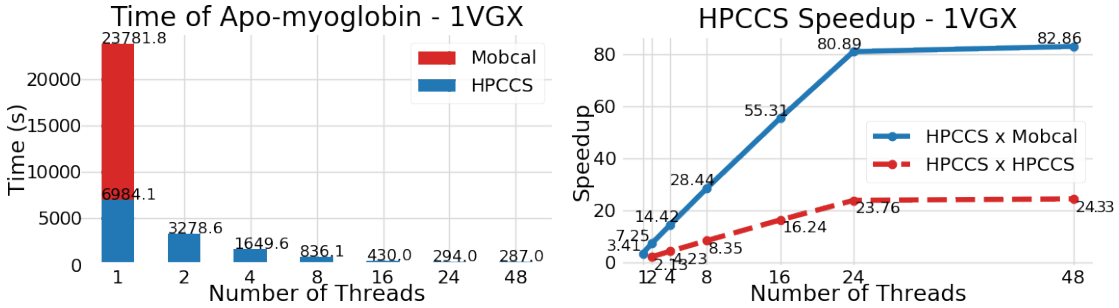

Figure 5.13: Time and Speedup of Apo-myoglobin - 1VGX with 2461 atoms.

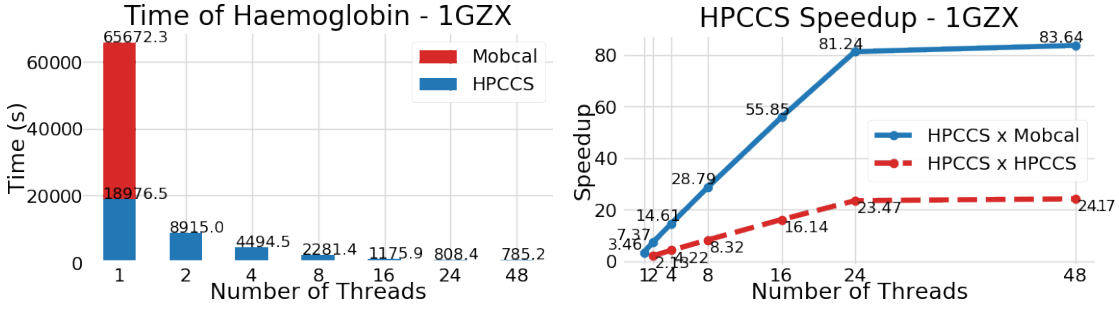

Figure 5.14: Time and Speedup of Haemoglobin - 1GZX with 4392 atoms.

As shown for the proteins from Perdita's group, Figures 5.1-5.14, time consuming reduces considerably when HPCCS is used, even in serial executions. As stated earlier, Mobcal was written at a time when there were only single-core computers, without using any

High Performance Computing technique. An important feature of the HPCCS program, that can be seen in these results, is its excellent speed up achieved using multi-cores processors. This allows users the efficient use of their desktops or clusters, running their calculations to get results in a viable time. Figure 5.15 is showing the relationship between the number of atoms and the execution time for Perdita's group.

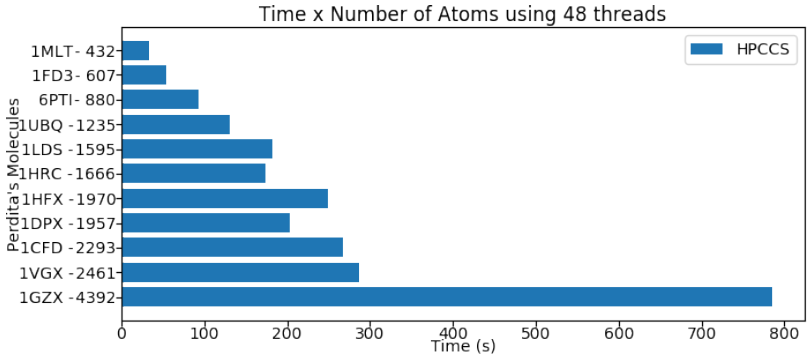

Figure 5.15: Perdita's molecules

It is expected that the greater the number of atoms, the greater the number of interactions between the gas and the ion that need to be computed, what results in an increasing the execution time. As shown in Figure 5.15, increasing the number of atoms has a considerable increase in simulation time, producing a 23 times increase for a protein with 10 times more atoms.

5.3 Bush's group

Figures 5.16-5.25 show the results for all proteins complexes from Bush's group.

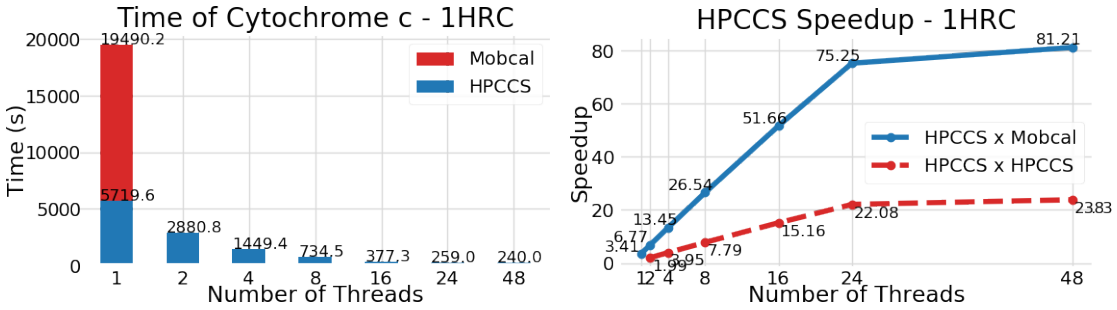

Figure 5.16: Time and Speedup of Cytochrome c - 1HRC with 1674 atoms.

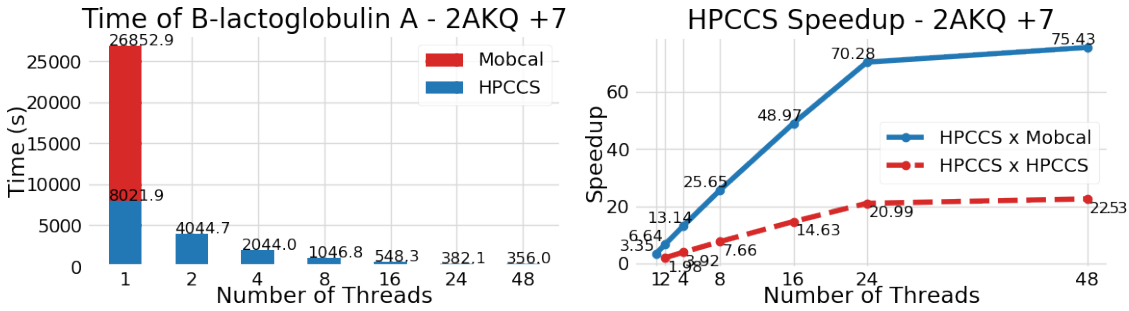

Figure 5.17: Time and Speedup of B-lactoglobulin - 2AKQ +7 with 2508 atoms.

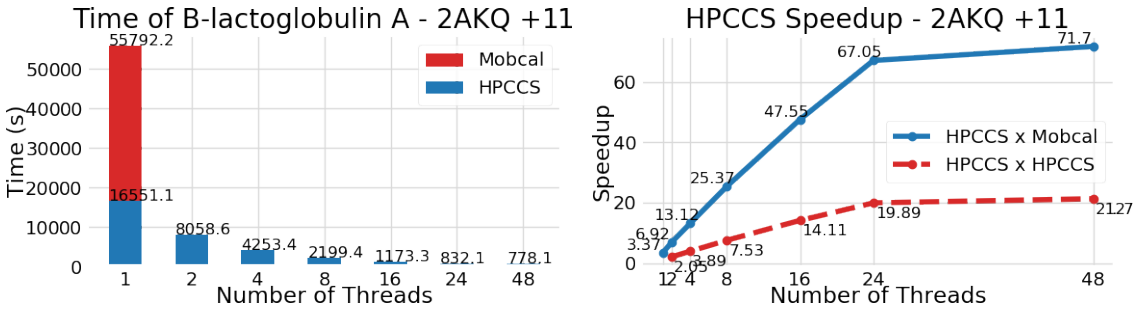

Figure 5.18: Time and Speedup of B-lactoglobulin - 2AKQ +11 with 5010 atoms.

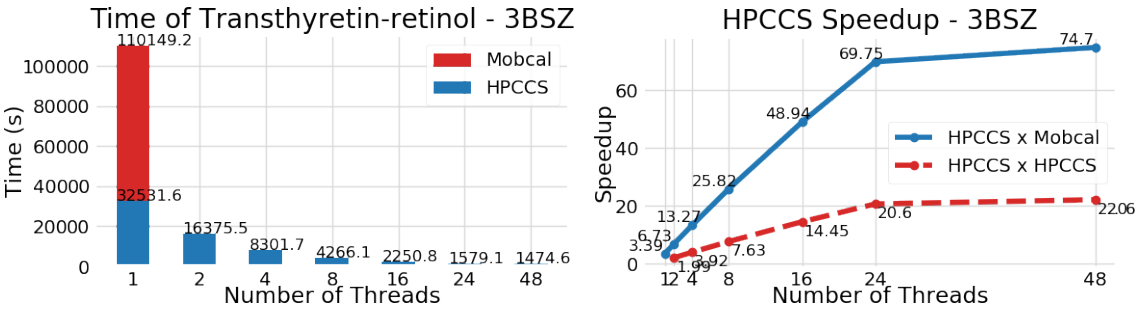

Figure 5.19: Time and Speedup of Transthyretin-retinol - 3BSZ with 7293 atoms.

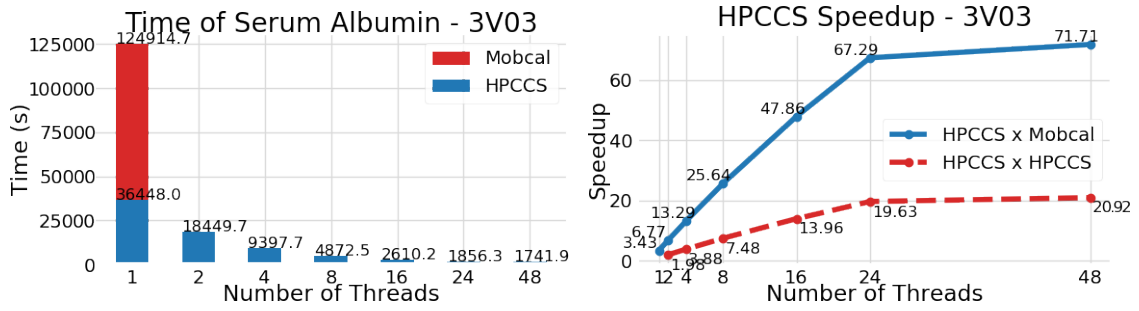

Figure 5.20: Time and Speedup of Serum Albumin - 3V03 with 9236 atoms.

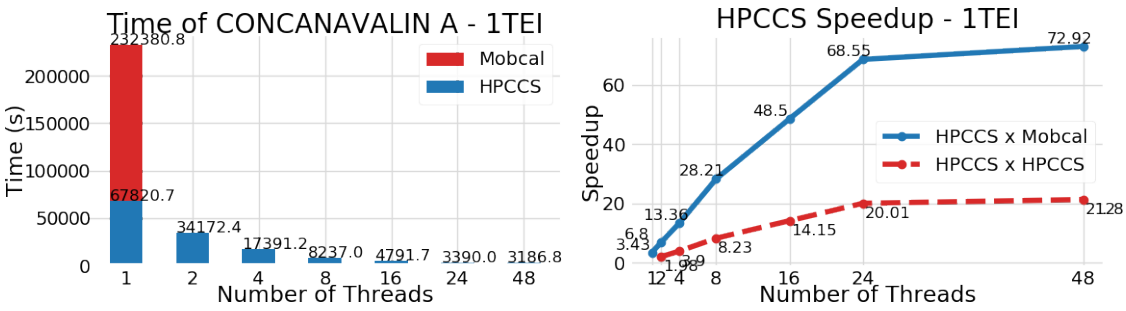

Figure 5.21: Time and Speedup of Concanavalin A - 1TEI with 14295 atoms.

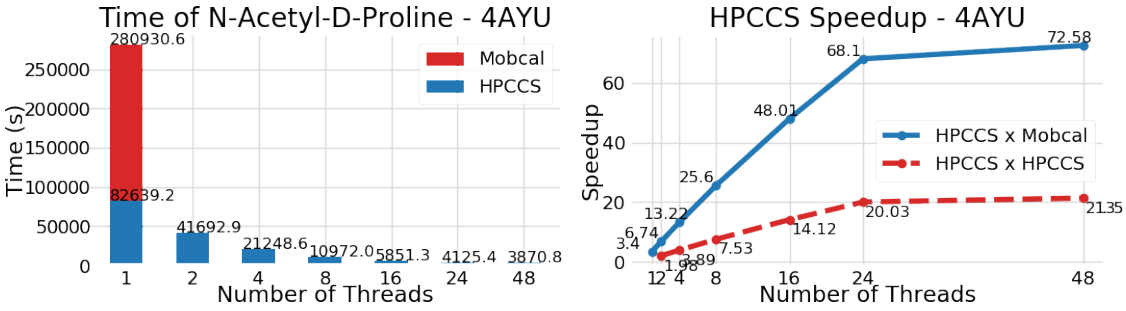

Figure 5.22: Time and Speedup of N-Acetyl-D-Proline - 4AYU with 16385 atoms.

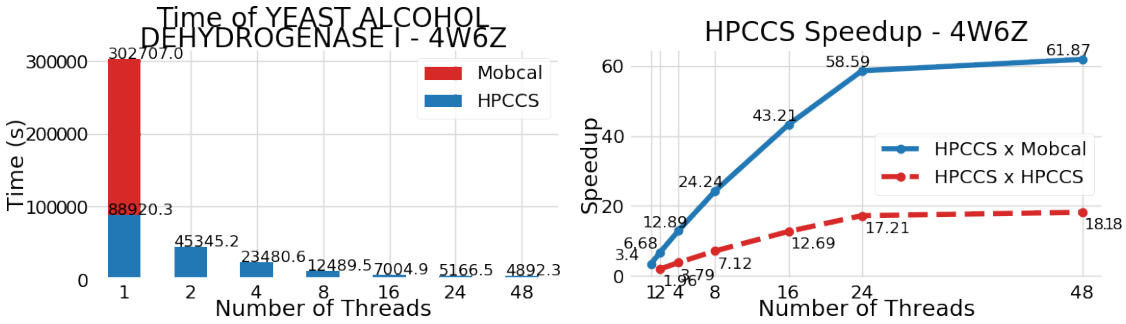

Figure 5.23: Time and Speedup of Yeast Alcohol Dehydrogenase I - 4W6Z with 8894 atoms.

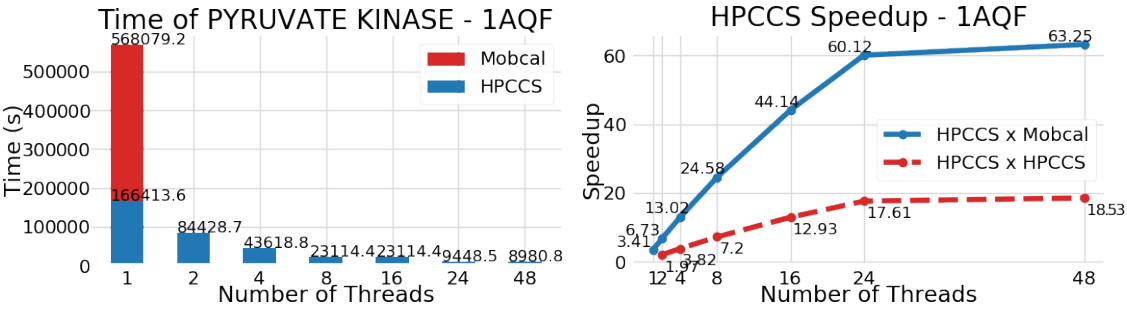

Figure 5.24: Time and Speedup of Pyruvate Kinase - 1AQF with 32170 atoms.

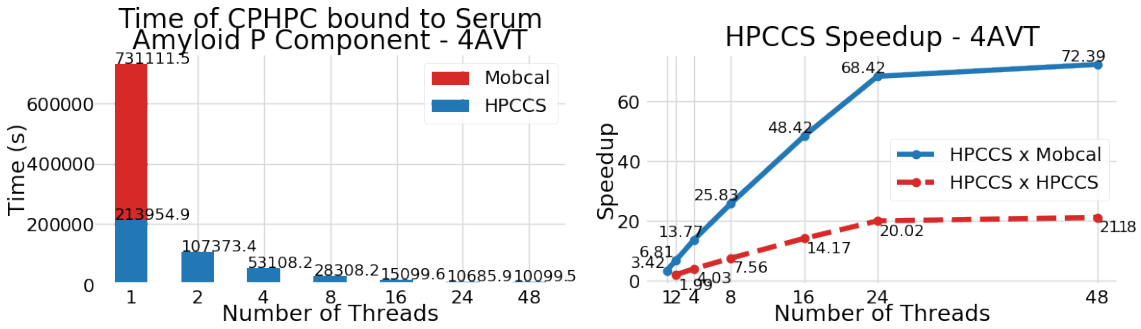

Figure 5.25: Time and Speedup of CPHPC bound to Serum Amyloid P Component - 4AVT with 32774 atoms.

The Bush's group includes protein complexes with a bigger number of atoms and more complicated shapes. The same tendency presented in Perdita's group can be observed herein. All the Mobcal calculations take 71x times longer to be finished, on average. For example, for our biggest protein complex 4AVT, while Mobcal took about 203 hours to process the calculation, the 48 threads HPCCS took 2,8 hours. Again, an excellent speedup was achieved. Figure 5.25 shows the relationship between the number of atoms and the execution time for Bush's group.

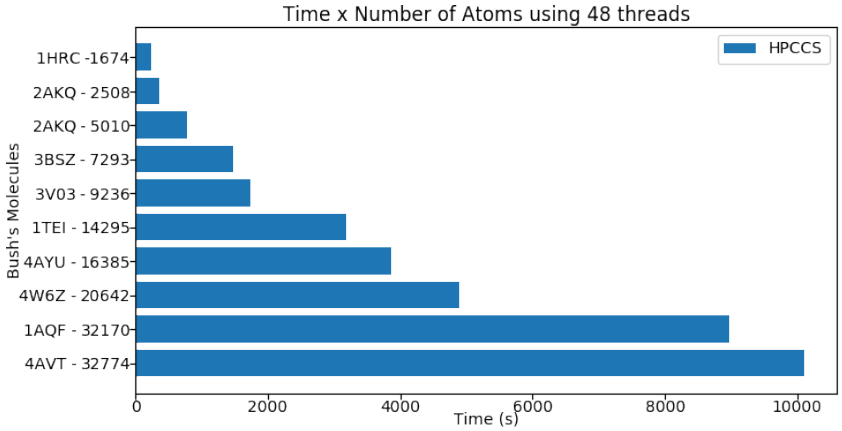

Figure 5.26: Bush's molecules

The same behavior as Perdita's group is shown for Bush's group. Increasing the number of atoms, the interactions between the gas increases too. More computation is needed to calculate the CCS, so the execution time will be increased. As can be seen from Figure 5.26, increasing the number of atoms tends to considerably increase execution, taking 42 times longer for a protein with 19,5 times more atoms.

5.4 Bush's group (MPI + OpenMP)

Figures 5.27-5.32, present the results for all proteins complexes from Bush's group using MPI + OpenMP version of HPCCS.

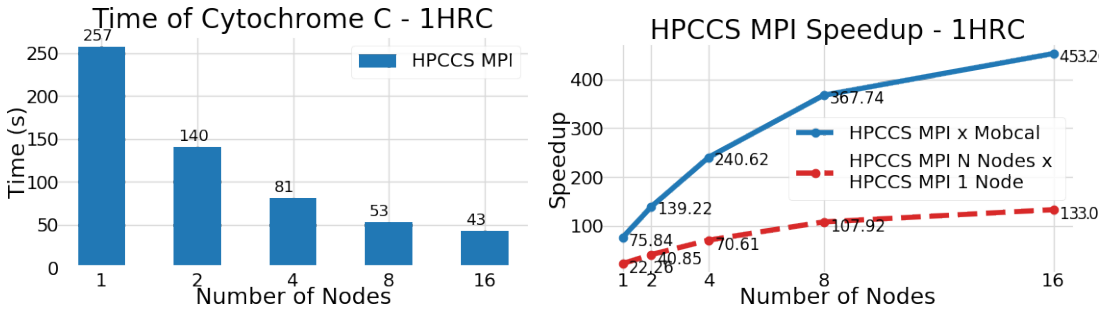

Figure 5.27: Time and Speedup of Cytochrome c - 1HRC with 1674 atoms.

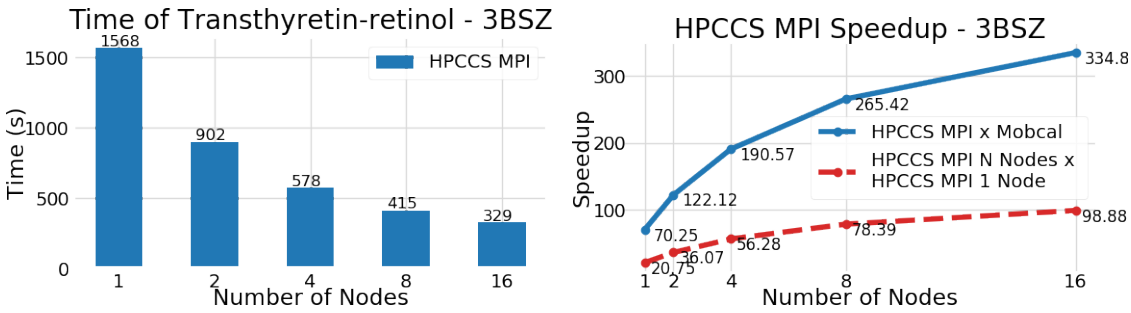

Figure 5.28: Time and Speedup of Transthyretin-retinol - 3BSZ with 7293 atoms.

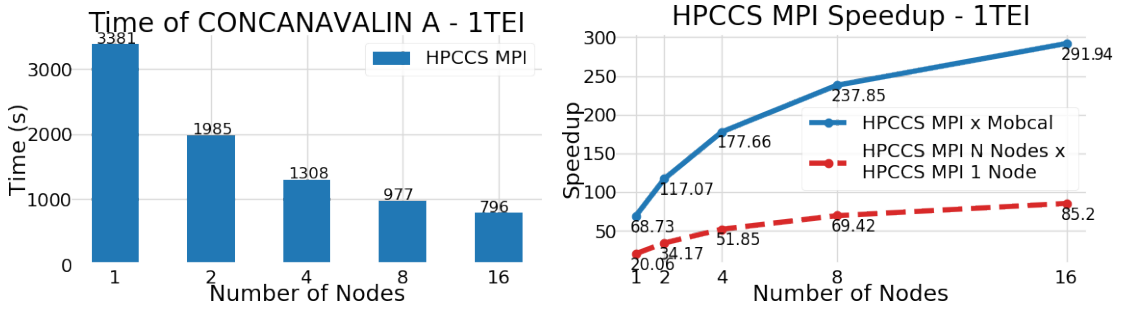

Figure 5.29: Time and Speedup of Concanavalin A - 1TEI with 14295 atoms.

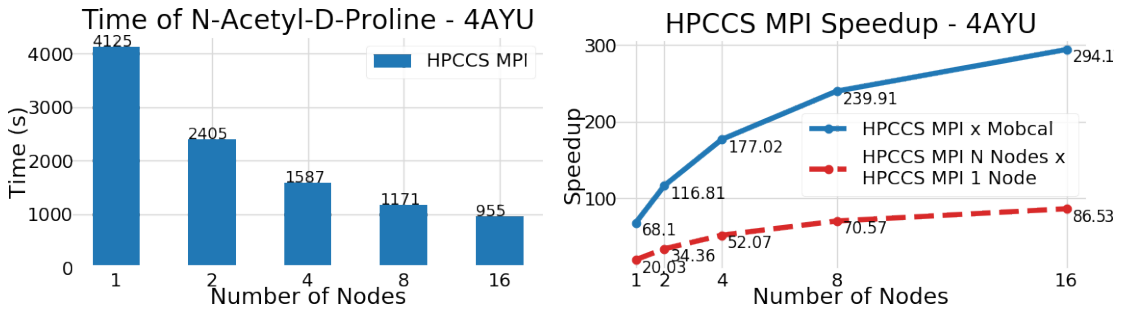

Figure 5.30: Time and Speedup of N-Acetyl-D-Proline - 4AYU with 16385 atoms.

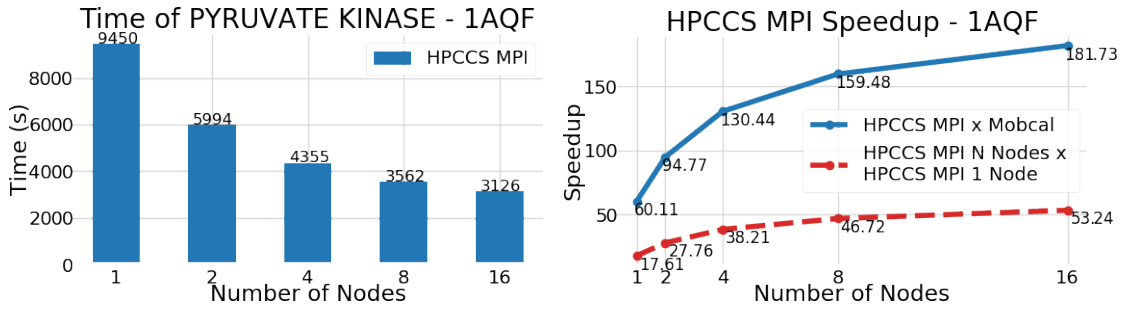

Figure 5.31: Time and Speedup of Pyruvate Kinase - 1AQF with 32170 atoms.

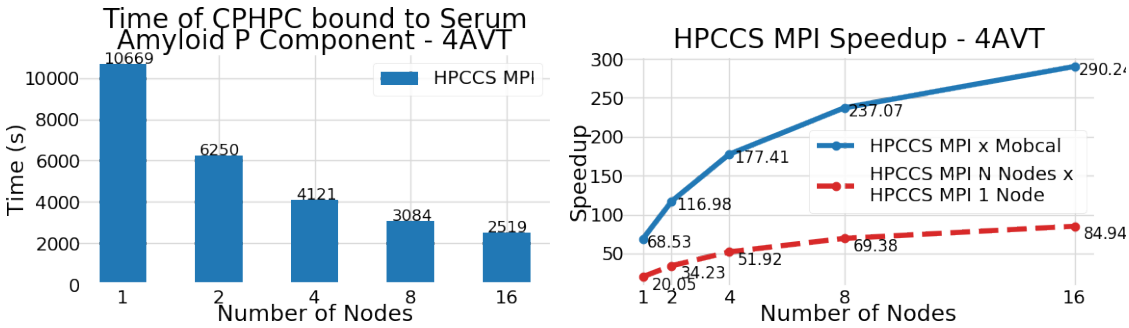

Figure 5.32: Time and Speedup of CPHPC bound to Serum Amyloid P Component - 4AVT with 32774 atoms.

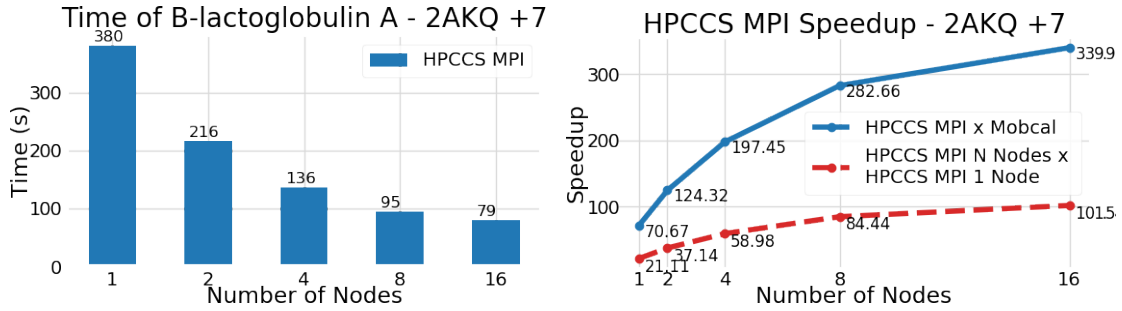

Figure 5.33: Time and Speedup of B-lactoglobulin - 2AKQ +7 with 2508 atoms.

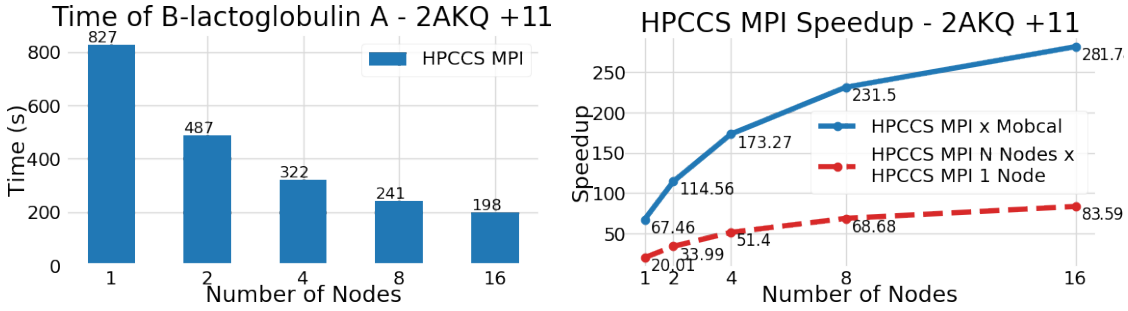

Figure 5.34: Time and Speedup of B-lactoglobulin - 2AKQ +11 with 5010 atoms.

5.5 HPCCS x Mobcal x IMPACT x Experimental Data

To compare the quality of our results against the experimental data, we present in Table 5.5 and 5.5 the results for Perdita's and Bush's groups, respectively, using the Mobcal, IMPACT and HPCCS programs. For the lack of space Table 5.5 omits the Mobcal results. Notice that the IMPACT program uses the projection approximation, a much simpler and faster algorithm.

PDB	**MW (Da)**	**Charge**	**Exp**	**Mobcal Dev.%**	**IMPACT Dev. %**	**HPCCS Dev. %**
1MLT	2846	+3	544	627 (15.3)	522 (4.0)	622 (14.3)
6PTI	6512	+4	770	922 (19.7)	750 (2.6)	917 (19.1)
1UBQ	8565	+4	1059	1054 (0.5)	855 (19.3)	1048 (1.0)
1HRC	12355	+3	1139	1323 (16.2)	1039 (8.8)	1328 (16.6)
		+5	1238	1324 (7.0)	1047 (15.4)	1328 (7.3)
1HFX	14178	+6	1342	1539 (14.7)	1219 (9.2)	1539 (14.7)
1DPX	14305	+5	1313	1484 (13.0)	1190 (9.4)	1485 (13.1)
		+6	1333	1485 (11.4)	1193 (10.5)	1488 (11.6)
		+8	1487	1491 (0.3)	1187 (20.2)	1493 (0.4)
1CFD	16700	+7	1900	2112 (11.2)	1688 (11.2)	2082 (9.6)
1VXG	17566	+8	1742	1738 (0.2)	1353 (22.3)	1721 (1.2)
1GZX	64447	+13	3051	4255 (39.5)	3200 (4.9)	2894 (5.2)
			Average ± SD	12.4 ±7	11.5 ±6	9.5 ±6

Table 5.1: Perdita's molecules, results between Mobcal, Impact and HPCCS

PDB	Chains	MW	Charge	Exp	IMPACT DEV. %	HPCCS DEV. %
2AKQ	1	18	+7	1660	1397 (15.8)	1784 (7.5)
2AKQ	2	37	+11	2850	2368 (16.9)	3069 (7.7)
3BSZ	4	56	+14	3410	2887 (15.3)	3751 (10.0)
1TEI	4	103	+20	5550	4687 (15.6)	5255 (5.3)
4AYU	5	125	+22	7030	5630 (19.9)	7514 (6.9)
1AQF	4	237	+31	10300	8788 (14.7)	11660 (13.2)
4AQF	10	250	+31	10400	8539 (17.9)	11550 (11.0)
				AVG	16.6±2	8.8±3

Table 5.2: Bush's molecules, results between Impact and HPCCS

It can be observed from Tables 5.5 and 5.5 that IMPACT underestimates the experimental values, while the errors from HPCCS, on average, slightly overestimate the CCS values, specially for N_2 gas. Further improvements in the force field parameters and/or minimization protocol could improve even more the accuracy of HPCCS. The PA algorithm ignores the long-range interactions and all details of the scattering process between the molecular ion and buffer gas, which is determinant to the conferred systematic error. Taken together, the results presented here indicates that HPCCS is capable of reproducing very well the experimental CCS values for a variety of molecular types, ranging from small proteins to protein complexes.

5.6 Discussions

The first and second groups, Perdita and Bush, are using only OpenMP for parallelization. For each simulation there are two charts, the first one shows the time spent on simulation and the second is the speedup. The solid blue line is the parallelized HPCCS speed-up when compared to sequential Mobcal and the red dashed line is comparing to sequential HPCCS, both with their serial version.

The sequential HPCCS is $\approx$ 3.5x faster than the Mobcal, as HPCCS is using many optimizations like vectorization and only the necessary amount of memory for each molecule simulation. Note that the same optimizations were used for its parallel version.

In the OpenMP version, the same execution behavior is showed, the bottom speedup is $\approx$ 62x using hyper-thread a technology that enables more than one thread per physical core resulting in a speed up of $\approx$ 83x when compared to Mobcal.

As show in the simulations the time of simulations increases as the number of atoms of molecular complexity increases. Increasing the number of physical cores the simulation time decreases. Comparing the HPPCS sequential version to Mobcal, the HPCCS speedup is ≈ 3.5x.

For the MPI + OpenMP version hype-thread is not used, only the physical cores available on each node. One MPI process triggers all available cores to process each simulation, considering each core to work on a single thread, which uses OpenMP to handle the execution.

There are two graphs on each experiment: The first graph shows the time from one to sixteen nodes. The second graph shows the speedup. The solid blue line is a comparison between one node and multiple nodes with on serial execution on Mobcal. The red dashed line is the comparison only with HPCS MPI with HPCCS Serial version.

Chapter 6

Conclusions

IM-MS techniques have become highly valued as a tool for (bio)chemical analysis and can be profitably used for both analyte separation and structural investigation of a wide range of sample types. Despite its enormous potential, IM-MS data interpretation is often challenging in a variety of different scenarios. For instance, when multiple molecular conformers influence the resulting collision cross section, when the surface roughness of the target molecule affects CCS (i.e., dependence on the buffer gas) or when the target is a large macromolecular system like protein complexes. In these cases, reliable and fast CCS estimates are needed to help in the interpretation of IM-MS data.

Mobcal is the most used software to calculate CCS and it uses three methods, PA, EHSS and TM. As described before, TM is the most accurate, as it considers the collision between the buffer gas and the molecule. But all this accuracy has a price, as discussed previously the TM method is expensive and requires many calculations, which is an obstacle for bigger molecules using Mobcal since its execution is entirely serial. This was a problem to solve for this area, and HPCCS was written based on Mobcal to solve it. To achieve that, it uses current HPC methods like modern processor architectures, code optimization, vectorization and other parallelization techniques.

Two versions of HPCCS were coded, one using only shared memory OpenMP computation on a single node and other which leverages on MPI + OpenMP to scale it on multiple nodes using distributed memory. The second version did not scale well using more than four nodes sue to the communication overhead between nodes.

HPCCS showed to be a good choice to calculate the CCS for bigger molecules, producing good results in a feasible times. A comparison to Mobcal was made using the processors available today in the market. HPCCS was able to process bigger molecules using multicore processors with vector units. Using a cluster with sixteen nodes with 24 cores each, the best speedup achieved was 453x when compared to Mobcal for 1HRC molecule with 1666 atoms.

Molecules can be calculate using the MPI + OpenMP version, on the other hand, even using a single desktop can compute CCS faster for small to medium molecules, than the traditional sequential Mobcal.

Chapter 7

Future Works

A carefull profiling of HPCCS showed that further performance can result if one manages to parallelize the b2max calculation using some DOACROSS parallelization algorithm. A new way to calculate it could be done to accelerate the calculation for bigger molecules. Some variables can be changed to float instead of double so the result will not differ to much from the original, while reducing execution time.

Other improvement can be the use of Linked Cell List (LCL) for the potential calculation. The current HPCCS calculates the iteration with all atoms which is not necessary, where the potential is zero or near to. Moreover the LCL it will calculate the potential only for those regions within a given cut radius.

To adapt HPCSS for LCL the code, it must be rewritten using other algorithms and considering a new algorithm to calculate CCS. Probably good speedups will be achieved using these future improvements.

Bibliography

[1] Christian Bleiholder, Thomas Wyttenbach, and Michael T. Bowers. A novel projection approximation algorithm for the fast and accurate computation of molecular collision cross sections (i). method. *International Journal of Mass Spectrometry*, 308(1):1–10, November 2011.

[2] Matthew F. Bush, Zoe Hall, Kevin Giles, John Hoyes, Carol V. Robinson, and Brandon T. Ruotolo. Collision cross sections of proteins and their complexes: A calibration framework and database for gas-phase structural biology. *Analytical Chemistry*, 82(22):9557–9565, November 2010.

[3] Iain Campuzano, Matthew F. Bush, Carol V. Robinson, Claire Beaumont, Keith Richardson, Hyungjun Kim, and Hugh I. Kim. Structural characterization of drug-like compounds by ion mobility mass spectrometry: Comparison of theoretical and experimentally derived nitrogen collision cross sections. *Analytical Chemistry*, 84(2):1026–1033, dec 2011.

[4] Leonardo Dagum and Ramesh Menon. Openmp: An industry-standard api for shared-memory programming. *IEEE Comput. Sci. Eng.*, 5(1):46–55, January 1998.

[5] Todd J. Dolinsky, Jens E. Nielsen, J. Andrew McCammon, and Nathan A. Baker. PDB2PQR: an automated pipeline for the setup of Poisson–Boltzmann electrostatics calculations. *Nucleic Acids Research*, $32(\text{suppl}_2) : W665 - -W667, 072004$.

[6] Message P Forum. Mpi: A message-passing interface standard. Technical report, Knoxville, TN, USA, 1994.

[7] Jürgen H. Gross. *Mass Spectrometry*. Springer Berlin Heidelberg, 2011.

[8] Jürgen H Gross. *Mass Spectrometry*. Springer International Publishing, 2017.

[9] Georg Hager and Gerhard Wellein. *Introduction to High Performance Computing for Scientists and Engineers*. CRC Press, Inc., Boca Raton, FL, USA, 1st edition, 2010.

[10] F. James. RANLUX: A fortran implementation of the high-quality pseudorandom number generator of lüscher. *Computer Physics Communications*, 79(1):111–114, February 1994.

[11] James Jeffers and James Reinders. *Intel Xeon Phi Coprocessor High Performance Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2013.

[12] Wentao Jiang and Renã A.S. Robinson. *Ion Mobility-Mass Spectrometry.* American Cancer Society, 2013.

[13] J. E. Jones. On the determination of molecular fields. II. from the equation of state of a gas. *Proceedings of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 106(738):463–477, October 1924.

[14] Ewa Jurneczko and Perdita E. Barran. How useful is ion mobility mass spectrometry for structural biology? the relationship between protein crystal structures and their collision cross sections in the gas phase. *Analyst*, 136:20–28, 2011.

[15] Elmar Krieger and Gert Vriend. New ways to boost molecular dynamics simulations. *Journal of Computational Chemistry*, 36(13):996–1007, 2015.

[16] Victor V. Laiko. Orthogonal extraction ion mobility spectrometry. *Journal of the American Society for Mass Spectrometry*, 17(4):500–507, apr 2006.

[17] Erik G. Marklund, Matteo T. Degiacomi, Carol V. Robinson, Andrew J. Baldwin, and Justin L.P. Benesch. Collision cross sections for structural proteomics. *Structure*, 23(4):791–799, April 2015.

[18] Stephen Marsland. *Machine Learning: An Algorithmic Perspective, Second Edition.* Chapman & Hall/CRC, 2nd edition, 2014.

[19] Makoto Matsumoto and Takuji Nishimura. Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Trans. Model. Comput. Simul.*, 8(1):3–30, January 1998.

[20] MF Mesleh, JM Hunter, AA Shvartsburg, George C Schatz, and MF Jarrold. Structural information from ion mobility measurements: effects of the long-range potential. *The Journal of Physical Chemistry*, 100(40):16082–16086, 1996.

[21] David Morin. *Introduction to Classical Mechanics: With Problems and Solutions.* Cambridge University Press, 2008.

[22] Phil Price. Standard definitions of terms relating to mass spectrometry. *Journal of the American Society for Mass Spectrometry*, 2(4):336–348, aug 1991.

[23] Reuven Y Rubinstein and Dirk P Kroese. *Simulation and the Monte Carlo method*, volume 10. John Wiley & Sons, 2016.

[24] Alexandre A. Shvartsburg, George C. Schatz, and Martin F. Jarrold. Mobilities of carbon cluster ions: Critical importance of the molecular attractive potential. *The Journal of Chemical Physics*, 108(6):2416–2423, feb 1998.

[25] Thomas Wyttenbach, Gert Helden, Joseph J. Batka, Douglas Carlat, and Michael T. Bowers. Effect of the long-range potential on ion mobility measurements. *Journal of the American Society for Mass Spectrometry*, 8(3):275–282, mar 1997.

[26] Leandro Zanotto, Gabriel Heerdt, Paulo C. T. Souza, Guido Araujo, and Munir S. Skaf. High performance collision cross section calculation-HPCCS. *Journal of Computational Chemistry*, 39(21):1675–1681, mar 2018.

[27] Roman A Zubarev, David M Horn, Einar K Fridriksson, Neil L Kelleher, Nathan A Kruger, Mark A Lewis, Barry K Carpenter, and Fred W McLafferty. Electron capture dissociation for structural characterization of multiply charged protein cations. *Analytical chemistry*, 72(3):563–573, 2000.

Appendix A

Attachment 1

The following tables are the results using OpenMP comparing to Serial Versions of Mobcal and HPCCS. NOTE: *Speedup HPCCS x Mobcal - **Speedup HPCCS Parallel x HPCCS Serial

Protein	**Mobcal**		**Number of Threads on HPCCS**					
1MLT	**Serial**	**Serial**	**2T**	**4T**	**8T**	**16T**	**24T**	**48T**
1	2652	752	373	188	96	51	35	34
2	2653	751	372	188	96	51	36	34
3	2654	751	373	188	96	51	35	34
4	2652	751	372	188	96	51	35	34
5	2654	751	373	188	96	51	36	34
6	2653	751	373	188	96	51	35	34
7	2654	752	372	189	96	51	35	34
8	2653	753	372	189	97	51	35	34
9	2654	752	372	188	96	51	35	34
10	2654	793	373	189	96	51	35	34
AVG	2653,30	755,70	372,50	188,30	96,10	51,00	35,20	34,00
Median	2653,50	751,50	372,50	188,00	96,00	51,00	35,00	34,00
DEV	0,82	13,12	0,53	0,48	0,32	0,00	0,42	0,00
Speedup*	1	3,51	7,12	14,09	27,61	52,03	75,38	78,04
Speedup**			2,03	4,01	7,86	14,82	21,47	22,23
Efficiency			1,01	1,00	0,98	0,93	0,89	0,46

Table A.1: Experiment Results using He buffer gas with Molecule 1MLT

Protein	**Mobcal**		**Number of Threads on HPCCS**					
1FD3	**Serial**	**Serial**	**2T**	**4T**	**8T**	**16T**	**24T**	**48T**
1	3604	1215	612	309	158	82	57	55
2	3605	1214	612	309	158	82	57	55
3	3603	1216	611	309	158	82	57	55
4	3604	1215	612	309	158	82	57	55
5	3623	1216	611	309	158	82	57	55
6	3604	1216	611	309	158	82	57	55
7	3604	1214	611	309	158	82	57	55
8	3604	1218	611	309	158	82	57	55
9	3622	1215	612	309	158	82	57	55
10	3605	1214	612	309	158	82	57	55
AVG	3607,80	1215,30	611,50	309,00	158,00	82,00	57,00	55,00
Median	3604,00	1215,00	611,50	309,00	158,00	82,00	57,00	55,00
DEV	7,772	1,252	0,527	0,000	0,000	0,000	0,000	0,000
Speedup*	1	2,97	5,90	11,68	22,83	44,00	63,29	65,60
Speedup**			1,99	3,93	7,69	14,82	21,32	22,10
Efficiency			0,99	0,98	0,96	0,93	0,89	0,46

Table A.2: Experiment Results using He buffer gas with Molecule 1FD3

Protein	**Mobcal**		**Number of Threads on HPCCS**					
6PTI	**Serial**	**Serial**	**2T**	**4T**	**8T**	**16T**	**24T**	**48T**
1	7702	2154	1076	541	274	141	96	94
2	7701	2151	1075	540	274	141	96	94
3	7700	2150	1076	540	274	141	96	94
4	7702	2151	1076	540	274	141	96	94
5	7703	2151	1075	541	274	141	96	94
6	7701	2153	1075	541	274	141	96	94
7	7701	2151	1075	540	274	141	96	94
8	7701	2153	1075	541	274	141	96	94
9	7702	2153	1075	540	274	141	96	94
10	7702	2151	1081	540	274	141	96	94
AVG	7701,50	2286,00	1075,90	540,40	274,00	141,00	96,00	94,00
Median	7701,50	2151,00	1075,00	540,00	274,00	141,00	96,00	94,00
DEV	0,850	1,317	1,853	0,516	0,000	0,000	0,000	0,000
Speedup*	1	3,37	7,16	14,25	28,11	54,62	80,22	81,93
Speedup**			2,12	4,23	8,34	16,21	23,81	24,32
Efficiency			1,06	1,06	1,04	1,01	0,99	0,51

Table A.3: Experiment Results using He buffer gas with Molecule 6PTI

Protein	**Mobcal**		**Number of Threads on HPCCS**					
1UBQ	**Serial**	**Serial**	**2T**	**4T**	**8T**	**16T**	**24T**	**48T**
1	10825	2990	1501	756	383	197	135	131
2	10824	2990	1501	756	383	197	135	131
3	10826	2990	1501	756	383	197	135	131
4	10824	2990	1501	756	383	197	135	131
5	10825	2991	1501	756	383	197	135	131
6	10825	3002	1501	756	383	197	135	131
7	10826	2990	1501	756	383	197	135	131
8	10826	2990	1502	756	383	197	135	131
9	10830	2990	1501	755	383	197	135	131
10	10827	2990	1501	755	383	197	135	131
AVG	10825,80	2991,30	1501,10	755,80	383,00	197,00	135,00	131,00
Median	10825,50	2990,00	1501,00	756,00	383,00	197,00	135,00	131,00
DEV	1,751	3,773	0,316	0,422	0,000	0,000	0,000	0,000
Speedup*	1	3,62	7,21	14,32	28,27	54,95	80,19	82,64
Speedup**			1,99	3,96	7,81	15,18	22,16	22,83
Efficiency			1,00	0,99	0,98	0,95	0,92	0,48

Table A.4: Experiment Results using He buffer gas with Molecule 1UBQ

Protein	**Mobcal**		**Number of Threads on HPCCS**					
1LDS	**Serial**	**Serial**	**2T**	**4T**	**8T**	**16T**	**24T**	**48T**
1	11460	4004	2012	1017	519	270	187	183
2	11459	4005	2012	1017	518	270	187	183
3	11463	4003	2012	1016	518	270	187	183
4	11451	4003	2012	1016	518	270	187	183
5	11452	4003	2014	1016	518	270	187	183
6	11460	4003	2012	1016	519	270	187	183
7	11459	4004	2012	1016	519	270	187	183
8	11458	4003	2012	1016	519	270	187	183
9	11460	4005	2012	1016	518	270	187	183
10	11460	4004	2012	1016	519	270	187	183
AVG	11458,20	4003,70	2012,20	1016,20	518,50	270,00	187,00	183,00
Median	11459,50	4003,50	2012,00	1016,00	518,50	270,00	187,00	183,00
DEV	3,765	0,823	0,632	0,422	0,527	0,000	0,000	0,000
Speedup*	1	2,86	5,69	11,28	22,10	42,44	61,27	62,61
Speedup**			1,99	3,94	7,72	14,83	21,41	21,88
Efficiency			0,99	0,98	0,97	0,93	0,89	0,46

Table A.5: Experiment Results using He buffer gas with Molecule 1LDS

Protein	**Mobcal**		**Number of Threads on HPCCS**					
1HRC	**Serial**	**Serial**	**2T**	**4T**	**8T**	**16T**	**24T**	**48T**
1	14016	3881	1951	983	500	258	178	174
2	14018	3882	1949	982	500	258	178	174
3	14019	3880	1949	983	500	258	178	174
4	14018	3881	1949	983	500	258	178	174
5	14023	3880	1948	983	500	261	178	174
6	14020	3881	1950	983	500	258	178	174
7	14027	3881	1950	983	500	258	178	174
8	14004	3882	1949	983	500	258	178	174
9	14018	3882	1949	983	500	258	178	174
10	14016	3882	1949	982	500	258	178	174
AVG	14017,90	3881,20	1949,30	982,80	500,00	258,30	178,00	174,00
Median	14018,00	3881,00	1949,00	983,00	500,00	258,00	178,00	174,00
DEV	5,915	0,789	0,823	0,422	0,000	0,949	0,000	0,000
Speedup*	1	3,61	7,19	14,26	28,04	54,27	78,75	80,56
Speedup**			1,99	3,95	7,76	15,03	21,80	22,31
Efficiency			1,00	0,99	0,97	0,94	0,91	0,46

Table A.6: Experiment Results using He buffer gas with Molecule 1HRC

Protein	**Mobcal**		**Number of Threads on HPCCS**					
1HRC	**Serial**	**Serial**	**2T**	**4T**	**8T**	**16T**	**24T**	**48T**
1	16405	4533	2278	1150	585	303	208	204
2	16402	4528	2278	1150	585	303	208	204
3	16403	4530	2279	1149	585	303	208	204
4	16401	4534	2280	1150	585	303	208	204
5	16403	4530	2279	1151	585	303	208	204
6	16405	4530	2278	1150	585	303	208	204
7	16403	4528	2278	1150	585	303	209	204
8	16405	4533	2277	1149	585	303	208	204
9	16402	4531	2278	1150	585	303	208	204
10	16401	4551	2279	1150	585	303	208	204
AVG	16403,00	4532,80	2278,40	1149,90	585,00	303,00	208,10	204,00
Median	16403,00	4530,50	2278,00	1150,00	585,00	303,00	208,00	204,00
DEV	1,563	6,713	0,843	0,568	0,000	0,000	0,316	0,000
Speedup*	1	3,62	7,20	14,26	28,04	54,14	78,82	80,41
Speedup**			1,99	3,94	7,75	14,96	21,78	22,22
Efficiency			0,99	0,99	0,97	0,93	0,91	0,46

Table A.7: Experiment Results using He buffer gas with Molecule 1HRC

Protein	**Mobcal**		**Number of Threads on HPCCS**					
1HRC	**Serial**	**Serial**	**2T**	**4T**	**8T**	**16T**	**24T**	**48T**
1	20426	5616	2819	1420	720	371	254	249
2	20427	5619	2818	1420	720	371	254	249
3	20453	5616	2818	1420	720	371	254	249
4	20418	5616	2819	1420	720	371	254	249
5	20456	5616	2819	1420	720	371	254	249
6	20417	5632	2819	1420	720	371	254	249
7	20416	5616	2820	1420	720	371	254	249
8	20415	5615	2819	1420	721	371	255	249
9	20419	5616	2819	1420	721	371	254	249
10	20432	5619	2819	1420	720	371	254	249
AVG	20427,90	5618,10	2818,90	1420,00	720,20	371,00	254,10	249,00
Median	20422,50	5616,00	2819,00	1420,00	720,00	371,00	254,00	249,00
DEV	15,074	5,065	0,843	0,568	0,000	0,000	0,316	0,000
Speedup**	1	3,64	7,25	14,39	28,36	55,06	80,39	82,04
Speedup**			1,99	3,96	7,80	15,14	22,11	22,56
Efficiency			1,00	0,99	0,98	0,95	0,92	0,47

Table A.8: Experiment Results using He buffer gas with Molecule 1HRC

Protein	**Mobcal**		**Number of Threads on HPCCS**					
1HFX	**Serial**	**Serial**	**2T**	**4T**	**8T**	**16T**	**24T**	**48T**
1	20426	5616	2819	1420	720	371	254	249
2	20427	5619	2818	1420	720	371	254	249
3	20453	5616	2818	1420	720	371	254	249
4	20418	5616	2819	1420	720	371	254	249
5	20456	5616	2819	1420	720	371	254	249
6	20417	5632	2819	1420	720	371	254	249
7	20416	5616	2820	1420	720	371	254	249
8	20415	5615	2819	1420	721	371	255	249
9	20419	5616	2819	1420	721	371	254	249
10	20432	5619	2819	1420	720	371	254	249
AVG	20427,90	5618,10	2818,90	1420,00	720,20	371,00	254,10	249,00
Median	20422,50	5616,00	2819,00	1420,00	720,00	371,00	254,00	249,00
DEV	15,074	5,065	0,843	0,568	0,000	0,000	0,316	0,000
Speedup*	1	3,64	7,25	14,39	28,36	55,06	80,39	82,04
Speedup**			1,99	3,96	7,80	15,14	22,11	22,56
Efficiency			1,00	0,99	0,98	0,95	0,92	0,47

Table A.9: Experiment Results using He buffer gas with Molecule 1HFX

Protein	**Mobcal**		**Number of Threads on HPCCS**					
1DPX	**Serial**	**Serial**	**2T**	**4T**	**8T**	**16T**	**24T**	**48T**
1	16421	4554	2291	1156	588	304	209	204
2	16411	4552	2291	1155	588	304	211	204
3	16413	4553	2291	1156	588	304	209	204
4	16413	4554	2291	1155	588	304	209	204
5	16410	4552	2291	1155	588	304	209	204
6	16410	4553	2291	1155	588	304	209	204
7	16413	4555	2291	1156	588	304	209	204
8	16410	4554	2291	1155	587	304	209	204
9	16410	4555	2291	1156	588	304	209	204
10	16418	4553	2291	1156	587	304	209	204
AVG	16412,90	4553,50	2291,00	1155,50	587,80	304,00	209,20	204,00
Median	16412,00	4553,50	2291,00	1155,50	588,00	304,00	209,00	204,00
DEV	3,784	1,080	0,568	0,000	0,422	0,000	0,316	0,000
Speedup*	1	3,60	7,16	14,20	27,92	53,99	78,46	80,46
Speedup**			1,99	3,94	7,75	14,98	21,77	22,32
Efficiency			0,99	0,99	0,97	0,94	0,91	0,47

Table A.10: Experiment Results using He buffer gas with Molecule 1DPX

Protein	**Mobcal**		**Number of Threads on HPCCS**					
1DPX	**Serial**	**Serial**	**2T**	**4T**	**8T**	**16T**	**24T**	**48T**
1	17880	4954	2490	1257	641	332	229	224
2	17884	4962	2493	1257	640	332	229	224
3	17883	4951	2492	1258	640	332	229	224
4	17885	4953	2490	1256	640	332	229	224
5	17882	4952	2491	1256	640	332	229	224
6	17877	4955	2490	1257	640	332	229	224
7	17881	4953	2491	1257	640	334	229	224
8	17880	4956	2492	1257	640	332	229	224
9	17886	4955	2492	1257	640	332	229	224
10	17882	4970	2494	1257	640	333	229	224
AVG	17882,00	4956,10	2491,50	1256,90	640,10	332,30	229,00	224,00
Median	17882,00	4954,50	2491,50	1257,00	640,00	332,00	229,00	224,00
DEV	2,667	5,744	1,354	0,568	0,316	0,675	0,000	0,000
Speedup*	1	3,61	7,18	14,23	27,94	53,81	78,09	79,83
Speedup**			1,99	3,94	7,74	14,91	21,64	22,13
Efficiency			0,99	0,99	0,97	0,93	0,90	0,46

Table A.11: Experiment Results using He buffer gas with Molecule 1DPX

Protein	**Mobcal**		**Number of Threads on HPCCS**					
1DPX	**Serial**	**Serial**	**2T**	**4T**	**8T**	**16T**	**24T**	**48T**
1	21401	5955	2981	1504	765	397	274	268
2	21404	5942	2980	1503	765	397	274	268
3	21404	5941	2980	1503	765	397	275	268
4	21400	5943	2980	1503	765	397	274	268
5	21404	5942	2982	1503	765	397	274	268
6	21399	5967	2980	1504	765	397	274	268
7	21401	5944	2981	1503	765	397	275	268
8	21401	5946	2981	1504	765	397	274	268
9	21398	5941	2981	1504	765	397	274	268
10	21431	5946	2981	1503	765	397	274	268
AVG	21404,30	5946,70	2980,70	1503,40	765,00	397,00	274,20	268,00
Median	21401,00	5943,50	2981,00	1503,00	765,00	397,00	274,00	268,00
DEV	9,615	8,247	0,675	0,516	0,000	0,000	0,422	0,000
Speedup*	1	3,60	7,18	14,24	27,98	53,92	78,06	79,87
Speedup**			2,00	3,96	7,77	14,98	21,69	22,19
Efficiency			1,00	0,99	0,97	0,94	0,90	0,46

Table A.12: Experiment Results using He buffer gas with Molecule 1DPX

Protein	**Mobcal**		**Number of Threads on HPCCS**					
1CFD	**Serial**	**Serial**	**2T**	**4T**	**8T**	**16T**	**24T**	**48T**
1	21379	5929	2976	1502	765	397	274	268
2	21364	5920	2975	1502	765	397	274	268
3	21365	5922	2979	1502	765	397	274	268
4	21380	5922	2976	1502	765	397	275	268
5	21375	5921	2975	1502	765	398	274	268
6	21371	5931	2978	1502	765	397	274	268
7	21368	5922	2975	1502	765	397	274	268
8	21377	5941	2975	1502	765	397	274	268
9	21365	5924	2975	1502	765	397	274	268
10	21401	5919	2976	1502	765	397	274	268
AVG	21374,50	5925,10	2976,00	1502,00	765,00	397,10	274,10	268,00
Median	21373,00	5922,00	2975,50	1502,00	765,00	397,00	274,00	268,00
DEV	11,078	6,773	1,414	0,000	0,000	0,316	0,316	0,000
Speedup*	1	3,61	7,18	14,23	27,94	53,83	77,98	79,76
Speedup**			1,99	3,94	7,75	14,92	21,62	22,11
Efficiency			1,00	0,99	0,97	0,93	0,90	0,46

Table A.13: Experiment Results using He buffer gas with Molecule 1CFD

Protein	**Mobcal**		**Number of Threads on HPCCS**					
1VGX	**Serial**	**Serial**	**2T**	**4T**	**8T**	**16T**	**24T**	**48T**
1	23777	6522	3279	1650	836	430	294	287
2	23776	6526	3277	1649	836	430	294	287
3	23775	6523	3277	1649	836	430	294	287
4	23774	6528	3278	1650	836	430	294	287
5	23778	6522	3279	1649	837	430	294	287
6	23781	6523	3279	1650	836	430	294	287
7	23779	6522	3277	1650	836	430	294	287
8	23815	6524	3279	1651	836	430	294	287
9	23779	6544	3280	1649	836	430	294	287
10	23784	6524	3281	1649	836	430	294	287
AVG	23781,80	6525,80	3278,60	1649,60	836,10	430,00	294,00	287,00
Median	23778,50	6523,50	3279,00	1649,50	836,00	430,00	294,00	287,00
DEV	12,026	6,680	1,350	0,699	0,316	0,000	0,000	0,000
Speedup*	1	3,64	7,25	14,42	28,44	55,31	80,89	82,86
Speedup**			1,99	3,96	7,81	15,18	22,20	22,74
Efficiency			1,00	0,99	0,98	0,95	0,92	0,47

Table A.14: Experiment Results using He buffer gas with Molecule 1VGX

Protein	**Mobcal**		**Number of Threads on HPCCS**					
1GZX	**Serial**	**Serial**	**2T**	**4T**	**8T**	**16T**	**24T**	**48T**
1	65680	17734	8914	4494	2280	1175	807	791
2	65684	17754	8926	4495	2281	1176	812	791
3	65657	17759	8911	4496	2281	1176	807	791
4	65652	17764	8916	4496	2282	1176	807	791
5	65647	17739	8909	4495	2281	1176	816	791
6	65682	17759	8917	4494	2282	1176	807	792
7	65683	17740	8914	4493	2282	1176	807	791
8	65681	17755	8911	4497	2283	1176	807	791
9	65674	17757	8911	4491	2282	1176	807	791
10	65683	17755	8921	4494	2280	1176	807	732
AVG	65672,30	17751,60	8915,00	4494,50	2281,40	1175,90	808,40	785,20
Median	65680,50	17755,00	8914,00	4494,50	2281,50	1176,00	807,00	791,00
DEV	14,469	10,135	5,249	1,716	0,966	0,316	3,098	18,695
Speedup*	1	3,70	7,37	14,61	28,79	55,85	81,24	83,64
Speedup**			1,99	3,95	7,78	15,10	21,96	22,61
Efficiency			1,00	0,99	0,97	0,94	0,91	0,47

Table A.15: Experiment Results using He buffer gas with Molecule 1GZX

Protein	**Mobcal**		**Number of Threads on HPCCS**					
1HRC	**Serial**	**Serial**	**2T**	**4T**	**8T**	**16T**	**24T**	**48T**
1	19495	5379	2700	1358	686	352	240	235
2	19487	5379	2697	1356	686	352	240	235
3	19488	5379	2697	1357	686	352	240	235
4	19482	5381	2698	1356	686	352	240	235
5	19491	5374	2697	1356	687	352	240	235
6	19487	5376	2696	1357	686	352	240	235
7	19501	5378	2697	1356	686	352	240	235
8	19490	5375	2697	1356	686	352	240	235
9	19495	5376	2697	1357	686	352	240	235
10	19486	5374	2697	1356	686	352	240	235
AVG	19490,20	5377,10	2697,30	1356,50	686,10	352,00	240,00	235,00
Median	19489,00	5377,00	2697,00	1356,00	686,00	352,00	240,00	235,00
DEV	5,514	2,424	1,059	0,707	0,316	0,000	0,000	0,000
Speedup*	1	3,62	7,23	14,37	28,41	55,37	81,21	82,94
Speedup**			1,99	3,96	7,84	15,28	22,40	22,88
Efficiency			1,00	0,99	0,98	0,95	0,93	0,48

Table A.16: Experiment Results using He buffer gas with Molecule 1HRC

Protein	**Mobcal**		**Number of Threads on HPCCS**					
2AKQ	**Serial**	**Serial**	**2T**	**4T**	**8T**	**16T**	**24T**	**48T**
1	26852	7499	3765	1900	967	501	346	338
2	26847	7491	3764	1900	967	501	346	338
3	26853	7491	3765	1900	967	501	346	338
4	26876	7488	3769	1901	967	501	346	338
5	26848	7488	3763	1900	967	501	346	338
6	26853	7488	3765	1900	967	501	346	338
7	26847	7501	3764	1900	967	505	346	338
8	26844	7492	3766	1900	967	501	346	338
9	26853	7490	3766	1900	967	501	346	338
10	26856	7493	3765	1900	967	501	346	338
AVG	26852,90	7492,10	3765,20	1900,10	967,00	501,40	346,00	338,00
Median	26852,50	7491,00	3765,00	1900,00	967,00	501,00	346,00	338,00
DEV	8,925	4,533	1,619	0,316	0,000	1,265	0,000	0,000
Speedup*	1	3,58	7,13	14,13	27,77	53,56	77,61	79,45
Speedup**			1,99	3,94	7,75	14,94	21,65	22,17
Efficiency			0,99	0,99	0,97	0,93	0,90	0,46

Table A.17: Experiment Results using He buffer gas with Molecule 2AKQ

Protein	**Mobcal**		**Number of Threads on HPCCS**					
2AKQ	**Serial**	**Serial**	**2T**	**4T**	**8T**	**16T**	**24T**	**48T**
1	55780	15492	7795	3942	2018	1057	736	721
2	55782	15511	7798	3946	2019	1058	736	721
3	55812	15506	7796	3946	2019	1057	736	721
4	55806	15486	7796	3945	2020	1057	737	721
5	55809	15479	7798	3945	2019	1057	737	721
6	55781	15482	7802	3945	2019	1057	739	721
7	55806	15478	7794	3944	2019	1059	736	721
8	55778	15494	7796	3944	2019	1057	736	721
9	55790	15478	7796	3944	2020	1057	736	721
10	55778	15495	7800	3946	2019	1057	736	721
AVG	55792,20	15490,10	7797,10	3944,70	2019,10	1057,30	736,50	721,00
Median	55786,00	15489,00	7796,00	3945,00	2019,00	1057,00	736,00	721,00
DEV	14,305	11,695	2,424	1,252	0,568	0,675	0,972	0,000
Speedup*	1	3,60	7,16	14,14	27,63	52,77	75,75	77,38
Speedup**			1,99	3,93	7,67	14,65	21,03	21,48
Efficiency			0,99	0,98	0,96	0,92	0,88	0,45

Table A.18: Experiment Results using He buffer gas with Molecule 2AKQ

Protein	**Mobcal**		**Number of Threads on HPCCS**					
3BSZ	**Serial**	**Serial**	**2T**	**4T**	**8T**	**16T**	**24T**	**48T**
1	110186	30390	15276	7702	3926	2039	1407	1380
2	110125	30377	15283	7704	3927	2036	1407	1380
3	110125	30361	15268	7704	3926	2037	1408	1380
4	110220	30381	15269	7704	3925	2038	1407	1380
5	110155	30355	15266	7702	3925	2036	1407	1380
6	110158	30368	15264	7706	3925	2036	1407	1380
7	110129	30370	15268	7708	3925	2036	1407	1380
8	110134	30369	15266	7705	3925	2037	1407	1380
9	110144	30369	15259	7704	3925	2036	1407	1380
10	110116	30373	15261	7703	3925	2036	1407	1380
AVG	110149,20	30371,30	15268,00	7704,20	3925,40	2036,70	1407,10	1380,00
Median	110139,00	30369,50	15267,00	7704,00	3925,00	2036,00	1407,00	1380,00
DEV	32,348	9,855	7,024	1,814	0,699	1,059	0,316	0,000
Speedup*	1	3,63	7,21	14,30	28,06	54,08	78,28	79,82
Speedup**			1,99	3,94	7,74	14,91	21,58	22,01
Efficiency			0,99	0,99	0,97	0,93	0,90	0,46

Table A.19: Experiment Results using He buffer gas with Molecule 3BSZ

Protein	**Mobcal**		**Number of Threads on HPCCS**					
3V03	**Serial**	**Serial**	**2T**	**4T**	**8T**	**16T**	**24T**	**48T**
1	124977	34259	17231	8716	4461	2334	1625	1593
2	124916	34226	17235	8714	4459	2334	1625	1593
3	124894	34213	17229	8714	4460	2334	1625	1594
4	124905	34244	17229	8712	4462	2334	1625	1593
5	124901	34214	17230	8718	4461	2334	1625	1593
6	124903	34208	17230	8713	4460	2334	1625	1593
7	124913	34213	17236	8718	4461	2334	1625	1593
8	124905	34226	17228	8715	4460	2334	1625	1593
9	124930	34244	17225	8716	4462	2334	1625	1593
10	124903	34232	17237	8712	4462	2334	1625	1593
AVG	124914,70	34227,90	17231,00	8714,80	4460,80	2334,00	1625,00	1593,10
Median	124905,00	34226,00	17230,00	8714,50	4461,00	2334,00	1625,00	1593,00
DEV	24,033	16,809	3,830	2,201	1,033	0,000	0,000	0,316
Speedup*	1	3,65	7,25	14,33	28,00	53,52	76,87	78,41
Speedup**			1,99	3,93	7,67	14,66	21,06	21,49
Efficiency			0,99	0,98	0,96	0,92	0,88	0,45

Table A.20: Experiment Results using He buffer gas with Molecule 3V03

Protein	**Mobcal**		**Number of Threads on HPCCS**					
1TEI	**Serial**	**Serial**	**2T**	**4T**	**8T**	**16T**	**24T**	**48T**
1	232384	63491	31923	16133	8242	4296	2985	2927
2	232333	63456	31917	16139	8242	4298	2985	2927
3	232397	63468	31915	16133	8242	4297	2989	2927
4	232489	63448	31912	16124	8239	4297	2983	2927
5	232355	63449	31915	16122	8241	4299	2984	2927
6	232340	63436	31920	16125	8241	4297	2985	2927
7	232392	63427	31919	16136	8241	4297	2987	2927
8	232477	63432	31911	16131	8243	4297	2984	2927
9	232320	63442	31908	16128	8238	4298	2986	2927
10	232321	63445	31912	16128	8242	4298	2985	2928
AVG	232380,80	63449,40	31915,20	16129,90	8237,00	4297,40	2985,30	2927,10
Median	232369,50	63446,50	31915,00	16129,50	8241,50	4297,00	2985,00	2927,00
DEV	60,780	18,798	4,614	5,466	1,524	0,843	1,703	0,316
Speedup*	1	3,66	7,28	14,41	28,21	54,07	77,84	79,39
Speedup**			1,99	3,93	7,70	14,76	21,25	21,68
Efficiency			0,99	0,98	0,96	0,92	0,89	0,45

Table A.21: Experiment Results using He buffer gas with Molecule 1TEI

Protein	**Mobcal**		**Number of Threads on HPCCS**					
4AYU	**Serial**	**Serial**	**2T**	**4T**	**8T**	**16T**	**24T**	**48T**
1	280940	77051	38837	19664	10043	5224	3619	3536
2	280944	76985	38844	19653	10041	5224	3619	3536
3	280890	77029	38833	19660	10041	5225	3621	3536
4	280924	77048	38882	19653	10046	5224	3620	3535
5	281007	77003	38872	19658	10045	5225	3620	3536
6	280953	76973	38847	19654	10042	5226	3618	3536
7	280953	77023	38849	19662	10041	5224	3619	3536
8	280882	76977	38830	19658	10043	5224	3618	3536
9	280929	76999	38842	19658	10042	5226	3618	3536
10	280884	77026	38836	19654	10041	5223	3618	3536
AVG	280930,60	77011,40	38847,20	19657,40	10042,50	5224,50	3619,00	3535,90
Median	280934,50	77013,00	38843,00	19658,00	10042,00	5224,00	3619,00	3536,00
DEV	38,494	28,175	16,976	3,864	1,780	0,972	1,054	0,316
Speedup*	1	3,65	7,23	14,29	27,97	53,77	77,63	79,45
Speedup**			1,98	3,92	7,67	14,74	21,28	21,78
Efficiency			0,99	0,98	0,96	0,92	0,89	0,45

Table A.22: Experiment Results using He buffer gas with Molecule 4AYU

Protein	**Mobcal**		**Number of Threads on HPCCS**					
4W6Z	**Serial**	**Serial**	**2T**	**4T**	**8T**	**16T**	**24T**	**48T**
1	302888	83038	41936	21384	11112	5982	4266	4195
2	302667	83021	41934	21384	11114	5976	4265	4195
3	302693	83006	41932	21383	11112	5976	4268	4195
4	302772	83006	41932	21382	11114	5976	4265	4195
5	302736	83063	41949	21386	11115	5976	4265	4195
6	302670	83016	41941	21385	11114	5977	4264	4194
7	302716	83009	41931	21385	11113	5978	4265	4195
8	302670	83010	41939	21381	11115	5977	4266	4164
9	302642	83015	41930	21385	11114	5975	4265	4195
10	302616	83003	41933	21385	11113	5978	4264	4195
AVG	302707,00	83018,70	41935,70	21384,00	11113,60	5977,10	4265,30	4191,80
Median	302681,50	83012,50	41933,50	21384,50	11114,00	5976,50	4265,00	4195,00
DEV	78,137	18,560	5,851	1,563	1,075	1,969	1,160	9,773
Speedup*	1	3,65	7,22	14,16	27,24	50,64	70,97	72,21
Speedup**			1,98	3,88	7,47	13,89	19,46	19,81
Efficiency			0,99	0,97	0,93	0,87	0,81	0,41

Table A.23: Experiment Results using He buffer gas with Molecule 4W6Z

Protein	**Mobcal**		**Number of Threads on HPCCS**					
4AQF	**Serial**	**Serial**	**2T**	**4T**	**8T**	**16T**	**24T**	**48T**
1	568075	155232	78209	39814	20629	11049	7847	7739
2	568276	155141	78201	39810	20635	11044	7846	7743
3	568070	155140	78185	39820	20636	11044	7849	7744
4	568024	155102	78182	39824	20638	11045	7848	7744
5	568114	155053	78196	39827	20634	11046	7847	7742
6	567898	155102	78176	39816	20634	11047	7846	7742
7	568189	155144	78184	39825	20633	11050	7845	7745
8	567994	155048	78184	39815	20631	11048	7846	7739
9	568082	155164	78196	39815	20637	11043	7847	7744
10	568070	155145	78192	39820	20635	11048	7848	7743
AVG	568079,20	155127,10	78190,50	39818,60	20634,20	11046,40	7846,90	7742,50
Median	568072,50	155140,50	78188,50	39818,00	20634,50	11046,50	7847,00	7743,00
DEV	102,919	54,098	10,069	5,502	2,700	2,366	1,197	2,068
Speedup*	1	3,66	7,27	14,27	27,53	51,43	72,40	73,37
Speedup**			1,98	3,90	7,52	14,04	19,77	20,04
Efficiency			0,99	0,97	0,94	0,88	0,82	0,42

Table A.24: Experiment Results using He buffer gas with Molecule 4AQF

Protein	**Mobcal**		**Number of Threads on HPCCS**					
4AVT	**Serial**	**Serial**	**2T**	**4T**	**8T**	**16T**	**24T**	**48T**
1	731441	199797	100291	50674	25879	13481	9354	9230
2	731338	199778	100284	50675	25876	13485	9356	9230
3	731002	199768	100280	50676	25875	13478	9356	9240
4	731108	199794	100283	50670	25872	13480	9356	9239
5	730930	199717	100286	50669	25874	13474	9357	9235
6	731035	199644	100294	50668	25875	13479	9355	9234
7	730944	199760	100281	50676	25876	13479	9353	9237
8	730984	199777	100281	50671	25878	13483	9357	9236
9	731173	199689	100283	50676	25874	13482	9356	9240
10	731160	199776	100307	50670	25873	13487	9354	9237
AVG	731111,50	199750,00	100287,00	50672,50	25875,20	13480,80	9355,40	9235,80
Median	731071,50	199772,00	100283,50	50672,50	25875,00	13480,50	9356,00	9236,50
DEV	170,332	50,337	10,069	5,502	2,700	2,366	1,197	2,068
Speedup*	1	3,66	7,29	14,43	28,26	54,23	78,15	79,16
Speedup**			1,99	3,94	7,72	14,82	21,35	21,63
Efficiency			1,00	0,99	0,96	0,93	0,89	0,45

Table A.25: Experiment Results using He buffer gas with Molecule 4AVT